REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED final 01 MAY 94 TO 31 OCT 95

4. TITLE AND SUBTITLE
FAULT-TOLERANCE IN DISTRIBUTED AND MULTIPROCESSOR REAL TIME SYSEMS

5. FUNDING NUMBERS
F49620-94-1-0276
2304/HS 61102F

6. AUTHOR(S)
DR. DHIRAJ K. PRADHAN

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
TEXAS A&M UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
COLLEGE STATION TX 77843-3112

AFOSR-TR-96
0203

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFSOR/NM
110 DUNCAN AVE SUITE B115
BOLLING AFB DC 20332-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
F49620-94-1-0276

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION UNLIMITED
APPROVED FOR PUBLIC RELEASE

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)
SEE REPORT FOR ABSTRACT

19960520 044

14. SUBJECT TERMS			15. NUMBER OF PAGES
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR

Fault-Tolerance in Distributed and Multiprocessor Real Time Systems

AFOSR F49620-94-1-0276

1 May 94 — 31 Oct 95

Final Report

Dhiraj K. Pradhan

pradhan@cs.tamu.edu
Phone: 409/862-2438
Fax: 409/862-2758

Department of Computer Science
Texas A&M University
College Station TX 77843-3112

Department of Computer Science
Texas A&M University

301 Harvey R. Bright Bldg • College Station, Texas 77843-3112

Table of Contents

1. "Fault Injection: A Method for Validating Computer-System Dependability," (J. Clark and D. Pradhan), *IEEE Computer*, June 1995.

2. "Providing Seamless Communications in Mobile Wireless Networks," (P. Krishna, B. Bakshi, N. Vaidya, and D. Pradhan), *lst International Conference on Mobile Computing and Networking*, November 1995.

3. "Performance and Reliability Assessment of I/O Subsystems," (F. Meyer, D. Pradhan and N. Vaidya), *4th IEEE International Workshop on Evaluation Techniques for Dependable Systems*, October 1995.

4. "Processor Allocation in Hypercube Multicomputers: Fast and Efficient Strategies for Cubic and Noncubic Allocation," (D. Das Sharma, D. Pradhan), *IEEE Transactions on Parallel and Distributed Systems*, October 1995.

5. "Cooperating Diverse Experts: A Methodology to Develop Quality Software for Critical Decision Support Systems," (D. Pradhan, H. Hecht, M. Hecht, F. Meyer and N. Vaidya), *IEEE Aerospace Applications Conference*, February 1995.

I. Research in Reliable I/O Design

1 Overview

In modern computing systems, I/O is a limiting factor in both performance and reliability. In this research investigation, we concentrated on evluating disk organizations that have been proposed in the literature to solve "the I/O problem".

The loss or unavailability of data may have substantial impact on various applications (finance, ground control, etc.). This often justifies using coding techniques to substantially lower the risk. For modest data spaces, the simple duplication of disks and controllers (disk duplexing) is an inexpensive and often effective solution. For systems requiring high I/O bandwidth or having vast data spaces, a fleet of disks will be needed. Redundant Arrays of Inexpensive Disks (RAID) [4, 6, 8] were proposed to overcome the reliability problems that come with relying on a large number of disks.

The classical RAID architecture consists of a set of disks attached to a custom controller as shown in Figure 1. The information on one of the disks is designated as parity while the remaining disks contain data [9]. The code implemented is the $(N+1, N)$ parity check code. It is suitable for correcting one erasure. An erasure is the equivalent of a self-identifying disk failure.

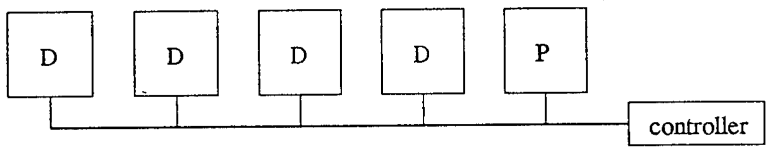


Figure 1: RAID Level 5 architecture

Since every data update involves writing both to the appropriate data disk and to the single parity disk, there may be a bottleneck at the parity disk. As a result, RAID Level 5 specifies that the parity be rotated. Imagine that the data placement is rotated one disk to the right. With $N+1$ disks, this may be done repeatedly to obtain $N+1$ different, but equally functional possibilities for parity. The data space is divided into $N+1$ smaller spaces of roughly equal size. Each one of these subdivisions is then associated with its own parity position. This distributes the read and write load evenly across the disks.

RAID Level 5 does a passable job of surviving disk failures. For I/O subsystems with more stringent reliability requirements or with many disks, RAID Level 6 is attractive. RAID Level 6 uses a $(N+2, N)$ Reed-Solomon code, which can correct any two disk failures.

We aimed specifically at assessing RAID-like disk organizations considering that other elements of the I/O subsystem can also fail—for example, controllers, cables, and power supplies. Current shipping RAID products are inherently unreliable, because their controllers are critical components. Since controllers can fail, it is natural to consider what happens when every disk attached to a controller becomes unavailable. So we view having multiple controllers with multiple disks connected to each; this looks logically like a two-dimensional array. If each column of the array has a controller assigned to it, then for simplicity we say that any failures in the I/O subsystem that do not merely affect a single disk affect a column of disks. Moreover, we will assume that the failure rates for such columns are not vanishingly small—so, unlike nearly all previous work, we do not ignore such failures. In referring to any failure affecting an entire column of disks, we will generally simply say 'controller failure'.

The classical two-dimensional arrangement to handle controller failures [10] is depicted in the left side of Figure 2. Each row of the array is a 'parity group' (is governed by one parity equation). The top row depicted, and all other rows, implements a $(N+1, N)$ code that can correct any single erasure. Each column of disks has a controller attached to it (two shown in the figure). Consequently, failure of a controller will affect one disk from each parity group—failure of two disks in the same parity group does not result from a controller failure. Reliability can be enhanced by providing a dual path to the disks, as is depicted in the left side of the figure. Two controller failures must occur in order to have the effect of making an entire column inaccessible.

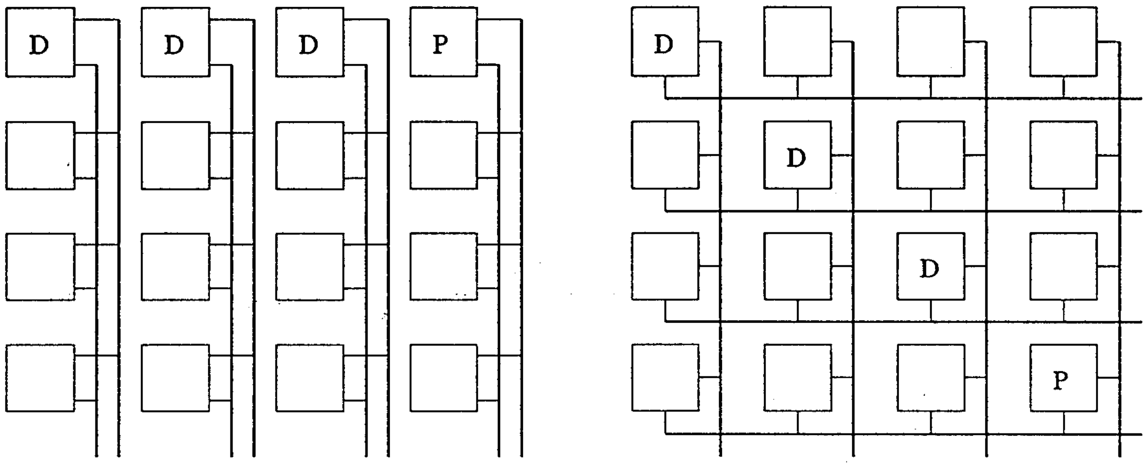


Figure 2: Dual path and crosshatch controller placements

We refer to these two options as 'single path' and 'dual path'. With the parity groups arranged horizontally, as shown, we refer to the method used as either 'single path horizontal' or as 'dual path horizontal' [10].

Another method, which is even more effective in protecting against controller failures, is the crosshatch [7] placement of controllers depicted in the right side of Figure 2. In the crosshatch arrangement, there are row controllers and column controllers. In order for a disk to become inaccessible from controller failures, both its corresponding row controller and its corresponding column controller must have failed. With this method, it is most sensible to have the parity groups comprised of diagonally related disks. This *crosshatch diagonal* method provides complete protection against triple controller failures. I.e., if three controllers fail, at most two disks may become inaccessible. Moreover, those two disks must be in either the same column or the same row—therefore, they cannot be in the same parity group.

We focus on the reliability of the proposed disk organizations. In the next section, we eliminate some proposed disk organizations, because they are subject to single points of failure. In the section following that, we describe our own proposed disk organization.

2 Disk Organizations and Their Masked Faults

It should be noted that the dual path horizontal and crosshatch diagonal methods depicted in Figure 2 require that each disk support access by two controllers. Such dual-ported disks are not commodity products to the extent that single-ported disks are. Therefore, we first classify RAID-like disk organizations according to whether they support single-ported disks or require dual-ported disks.

Table 1 shows which fault combinations are masked by various disk organizations. All of these disk organizations support single-ported disks. The column headings signify the component failures—e.g, 'DDC' means two disk failures and one controller failure. A method is awarded a '1' if it tolerates all combinations of faults of the quantity given. (E.g., RAID Level 5 tolerates all single disk failures, 'D'.) A method is awarded a '< 1' if it tolerates nearly all such combinations of failures. (E.g., disk duplexing is not guaranteed to tolerate all double disk failures, 'DD', but it tolerates nearly all of them.) '0' indicates the method does poorly with the fault combination. A dash signifies that the method already fails to tolerate a lesser fault combination. (E.g., no indication is given for disk duplexing and triple disk failures, because disk duplexing does not mask all double disk failure combinations.)

Method	D	C	DD	DC	CC	DDD	DDC	DCC	CCC
Disk duplexing	1	1	< 1	0	0	–	–	–	–
RAID Level 5 [6]	1	0	0	–	–	–	–	–	–
RAID Level 6 [6]	1	0	1	–	–	0	–	–	–
EVENODD [1]	1	0	1	–	–	0	–	–	–
Single path horizontal [10]	1	1	0	0	0	–	–	–	–
Single path ORAID GF(4)	1	1	1	0	0	< 1	–	–	–
Modified EVENODD	1	1	1	1	1	< 1	0	0	0

Table 1: Fault coverage using single-ported disks

Since RAID Level 5, RAID Level 6, and EVENODD do not tolerate all single points of failure, we will not discuss them further.

The disk organization given as *modified* EVENODD results from looking at the work in [1] from the perspective of unreliable controllers. That work, in discussing disk failures, represents each disk as a column—each element of the column being some portion of the disk. The code given in [1] does not have a high rate, so it is able to mask double column erasures. For the discussion in [1], that corresponds to double disk failures. But for our discussion, we can imagine that the code is implemented across multiple controllers—where each controller is a column and each disk is a column element. The strength of the code yields excellent fault masking. The penalties for this are: (1) the rate of the code is less, so more disks are needed, although the rate is higher than with disk duplexing and (2) writing to the file system requires more disk operations.

Table 2 shows the same information for disk organizations that require dual-ported disks. In this case the EVENODD disk organization is as in [1] and the controller is duplicated.

Since these disk organizations require dual-ported disks they are only plausible for ultra-reliable systems. As we discovered in our reliability analysis, only the crosshatch ORAID

Method	D	C	DD	DC	CC	DDD	DDC	DCC	CCC
Dual path RAID Level 5	1	1	0	1	0	–	–	–	–
Dual path RAID Level 6	1	1	1	1	0	0	1	–	–
Dual path EVENODD	1	1	1	1	0	0	1	–	–
Dual path horizontal [10]	1	1	0	1	1	–	–	0	1
Crosshatch diagonal [7]	1	1	0	1	1	–	–	0	1
Crosshatch ORAID GF(4)	1	1	1	1	1	< 1	1	1	1

Table 2: Fault coverage using dual-ported disks

disk organization can justify itself.

3 Description of ORAID Architecture

Figure 3 shows the basic physical architecture of ORAID. In this figure the 'single path' disk arrangement is shown.

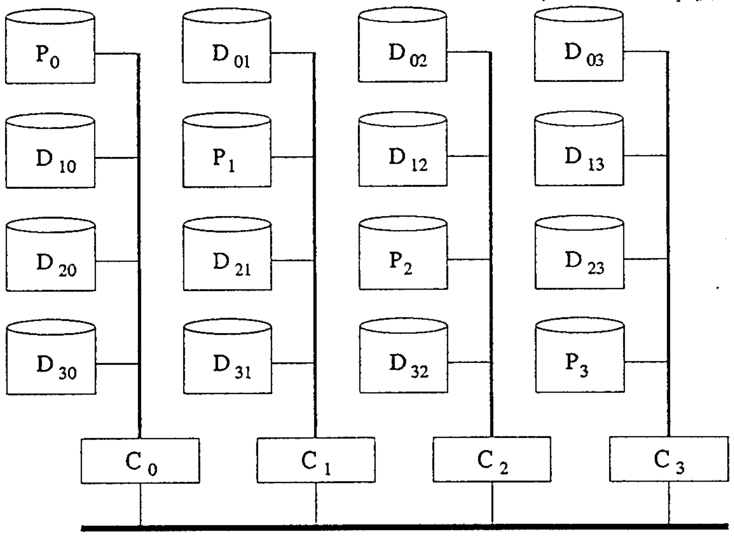


Figure 3: ORAID architecture

The parity disks—P_0, P_1, and so forth—are located along the major diagonal of the array. For convenience, we use the notation P_i and D_{ii} interchangeably. The controllers, C_i, are attached to each column.

Let there be k controllers and k^2 disks. The code used is systematic, so for each disk, D_{ij}, $i \neq j$, the disk simply contains the data. To read data, all that need be done is to determine which data disk contains the desired block.

The contents of the parity disks are defined as follows:

$$P_i = \sum_{j \neq i} D_{ji} + \alpha \sum_{j \neq i} D_{ij}$$

where addition is over a Galois field GF(2^p) for some positive integer p. α is a generator for GF(2^p). In the case of GF(2), $\alpha = 1$. There are several plausible choices for the Galois field: GF(2), GF(4), GF(8), GF(16), etc. We refer to these disk organizations as, for example, ORAID over GF(8). When constrained to allow single-ported disks, ORAID cannot really compete with the modified EVENODD disk organization. So we will only consider ORAID for systems with dual-ported disks (the crosshatch layout). To achieve an ultrareliable I/O subsystem requires, at the outset that all double disk failures be tolerated. Since ORAID over GF(2) is vulnerable to some double disk failures, we eliminate that as well. That leaves GF(4) or a higher-order field. Using a field of order greater than 8 does not allow ORAID to mask any additional fault combinations, so the natural possibilities are ORAID over GF(4) and ORAID over GF(8). (If not using GF(4), it will likely be appropriate to choose a field like GF(16), so that each symbol can be represented by 4 bits instead of 3.)

The code used is merely a parity product code with the row and column parities folded into each other. The diagonal elements of the product code are not used for data and the parity elements are positioned in their place.

As with all disk organizations we investigated, we assumed that the parities are horizontally rotated in order to balance the load across the disks under fault-free operation. For convenience in the ensuing discussion, we do not refer to parity rotation further. The analysis we conducted showed that taking parity rotation into consideration does not materially affect the results.

Applications make read and write requests of the data space. But in RAID architectures these requests do not map directly to disk read and write operations. We define five elemental I/O subsystem operations:

R	Read a disk block
W	Write a disk block
M	Perform a read-modify-write sequence on a disk block[6]
S	Scale a buffer block (i.e., multiply by a Galois field element)
A	Add (XOR) two buffers (from different disk controllers)

The scaling (S) and adding (A) operations are conducted by either the I/O processor or a custom disk controller; scaling in ORAID is only applicable when not using GF(2). In GF(2^p), addition may be made equivalent to XOR. So the read-modify-write operation consists of reading a disk block, XOR-ing it into the contents of a buffer, and then writing that buffer back into the place of the original disk block. The read-modify-write operation entails only one disk seek, but one additional disk revolution must occur instead of a disk read operation (assuming that a block is confined to one disk track). A read immediately followed by a write to the same disk block is considered to be a read-modify-write operation even when no modification (XOR) occurs or when the XOR result is not the data written.

Table 3 shows the total disk load under fault-free conditions. r and w are the rates of logical disk reads and writes, respectively. These logical reads and writes must be translated into the elemental I/O operations. Single path and dual path organizations have the same load.

Like RAID Level 6, ORAID requires updating three disks whenever a block of data is written. Both those schemes tolerate all double disk failures; and, as is well known, three

Method	Read	Write	Read-Modify-Write	Scale
Disk duplexing	r	$2w$	0	0
RAID Level 5 (3 disks)	$r+w$	$2w$	0	0
RAID Level 5	r	0	$2w$	0
RAID Level 6	r	0	$3w$	w
EVENODD	r	0	$3w$	0
Horizontal (single/dual)	r	0	$2w$	0
Modified EVENODD	r	0	$4w$	0
Crosshatch diagonal	r	0	$2w$	0
ORAID GF(4)	r	0	$3w$	w

Table 3: Fault-free disk load

updates are necessary to do that. The extra read-modify-write operation is the price that must be paid to tolerate all double disk failures.

To establish the claim that ORAID needs only one scaling operation and three read-modify-write disk operations for each data write request, one need only examine Figure 4.

4 Summary of Conclusions

We investigated RAID-like disk organizations proposed in the literature, with particular emphasis on their reliability when subject to controller failures. Since disk duplexing was the only viable disk organization for single-ported disks, we proposed two new disk organizations to achieve ultrareliable I/O subsystems in the presence of controller failures. The first method is a direct adaptation of the EVENODD disk organization propounded in [1] and achieves high reliability using commodity single-ported disks. The other disk organization introduced, crosshatch ORAID, achieves ultra high reliability using dual-ported disks.

Disk duplexing will be the preferred solution for most low-end file servers. It may be inadequate on either performance or reliability grounds. In both cases, the modified EVENODD disk organization is attractive. RAID Level 5 and RAID Level 6 are not comparable for either performance or reliability. While the single path horizontal disk organization [10] has the potential for performance comparable to the modified EVENODD disk organization, its reliability is seen to be uniformly even worse than disk duplexing.

When very high reliability is required, it may be necessary to use dual-ported disks to achieve this. (There is a coincidental opportunity to improve performance in doing so.) In considering the reliability of various dual-ported disk organizations, it is seen that only the crosshatch ORAID disk organization distinguishes itself from disk duplexing in terms of reliability. A more interesting comparison, considering the complexities involved, is whether the modified EVENODD disk organization is adequate, because it would allow the use of single-ported disks (albeit more of them). We can conclude that the modified EVENODD disk organization is suitable for ultrareliable I/O subsystems ($< 0.1\%$ probability of data

```
Determine $(i, j)$
$C_j(\text{buffer}_1) \leftarrow \text{Data}$

$C_j$:  $\text{buffer}_2 \leftarrow \text{buffer}_1 \oplus D_{ij}$
        —    buffer two contains modification
$C_j$:  $D_{ij} \leftarrow \text{buffer}_1$
        —    data now in place

$C_i(\text{buffer}_2) \leftarrow \alpha C_j(\text{buffer}_2)$
        —    $C_i$ now knows the modification (scaled)

$C_j$:  $\text{buffer}_1 \leftarrow \text{buffer}_2 \oplus P_j$
$C_j$:  $P_j \leftarrow \text{buffer}_1$
        —    parity updated

$C_i$:  $\text{buffer}_1 \leftarrow \text{buffer}_2 \oplus P_i$
$C_i$:  $P_i \leftarrow \text{buffer}_1$
        —    parity updated
```

Figure 4: Writing to D_{ij}

loss in any given year) provided *all* of the following are true: 1) controllers (data paths) are reasonably reliable, 2) disk reconstruction times (mostly disk size and speed) are reasonable, and 3) the number of data disks (disk size and data space size) are not gargantuan. Details are in the appendix.

References

[1] M. Blaum, J. Brady, J. Bruch, and J. Menon, "EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures," *IEEE Transactions on Computers,* vol. 44, no. 2, February 1995, pp. 192–202.

[2] W. Burkhard and J. Menon, "Disk Array Storage System Reliability," *23rd Fault-Tolerant Computing Symposium,* pp. 432–441, June 1993.

[3] R. Butler and S. Johnson, "Techniques for Modeling the Reliability of Fault-Tolerant Systems with the Markov State-Space Approach," *NASA Reference Publication 1348,* September 1995.

[4] J. Chandy and P. Banerjee, "Reliability Evaluation of Disk Array Architectures," *International Conference on Parallel Processing,* Saint Charles IL, pp. 264–269, August 1993.

[5] A. Drapeau, et alii, "RAID-II: A High-Bandwidth Network File Server," *21st International Symposium on Computer Architecture,* Chicago, pp. 234–244, April 1994.

[6] G. Gibson, *Redundant Disk Arrays: Reliable, Parallel Secondary Storage,* Ph.D. dissertation, MIT, 1992.

[7] S. Ng, "Crosshatch Disk Array for Improved Reliability and Performance," *21st International Symposium on Computer Architecture,* Chicago, pp. 255–264, April 1994.

[8] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *ACM SIGMOD Conference,* Chicago, pp. 109–116, June 1988.

[9] T. Rao and E. Fujiwara, *Error-Control Coding for Computer Systems,* Prentice Hall: Englewood Cliffs NJ, 1989.

[10] M. Schulze, G. Gibson, R. Katz, and D. Patterson, "How Reliable Is a RAID," *COMPCON,* San Francisco, pp. 118–123, February 1989.

[11] E. Schwabe and I. Sutherland, "Improved Parity-Declustered Layouts for Disk Arrays," *ACM Symposium on Parallel Algorithms and Architectures,* Cape May NJ, pp. 76–84, June 1994.

5 Publications Supported

The following attached publications were supported by AFOSR F49620–94–1–0276.

1. J. Clark and D. Pradhan, "Fault injection: A method for validating computer-system dependability," *IEEE Computer,* June 1995.

2. P. Krishna, B. Bakshi, N. Vaidya, and D. Pradhan, "Providing Seamless Communications in Mobile Wireless Networks," *1st International Conference on Mobile Computing and Networking,* November 1995.

3. F. Meyer, D. Pradhan, and N. Vaidya, "Performance and Reliability Assessment of I/O Subsystems," *4th IEEE International Workshop on Evaluation Techniques for Dependable Systems,* October 1995.

4. D. Das Sharma and D. Pradhan, "Processor allocation in hypercube multicomputers: Fast and efficient strategies for cubic and noncubic allocation," *IEEE Transactions on Parallel and Distributed Systems,* October 1995.

5. D. Pradhan, H. Hecht, M. Hecht, F. Meyer, and N. Vaidya, "Cooperating Diverse Experts: A Methodology to Develop Quality Software for Critical Decision Support Systems," *IEEE Aerospace Applications Conference,* February 1995.

A Reconstruction with ORAID

Let us define S_i, where $i \in [0, k-1]$, as the syndrome for parity equation i. Syndrome S_i is equal to the linear combination of the elements of the i-th row and the i-th column—as dictated by the parity equation—except that any erasures are not included. For example, parity equation i is

$$0 = \sum_j D_{ji} + \alpha \sum_{j \neq i} D_{ij}$$

If the contents of D_{hi} are unreliable, then

$$\begin{aligned} S_i &= \sum_{j \neq h} D_{ji} + \alpha \sum_{j \neq i} D_{ij} \\ &= \sum_j D_{ji} + \alpha \sum_{j \neq i} D_{ij} + D_{hi} \\ &= D_{hi} \end{aligned}$$

As seen above, $S_i = D_{hi}$. So we may obtain the missing contents of D_{hi} by computing syndrome S_i. Similarly, if D_{ih} contains unreliable data, with $h \neq i$, then

$$\begin{aligned} S_i &= \sum_j D_{ji} + \alpha \sum_{j \neq i,h} D_{ij} \\ &= \sum_j D_{ji} + \alpha \sum_{j \neq i} D_{ij} + \alpha D_{ih} \\ &= \alpha D_{ih} \end{aligned}$$

Again, we can obtain D_{ih}, because $D_{ih} = \alpha^{-1} S_i$. So ORAID tolerates any single disk failure.

Looking more closely at the syndrome equation, we see that each syndrome, S_i, is a linear combination of precisely the fault-free data from all disks of the form D_{ij} and D_{ji}. This leads immediately to the conclusion that a column failure is always tolerated. If a single column—say D_{0j}, D_{1j}, $\ldots$, $D_{(k-1)j}$—becomes inaccessible, the data may always be recovered. Specifically, for all $i \neq j$, D_{ij} can be obtained from syndrome S_i. After the data on all the other disks in the failed column is known, then D_{jj} (P_j) can be trivially recovered as well.

This observation about S_i also gives an immediate understanding of which double disk failures might not be correctable. If D_{ij} is one of the failed disks, then necessarily $i \neq j$. Otherwise, the other failed disk would have a unique subscript, its data could be reconstructed, and we would then be left with the case of a single disk failure. So let $i \neq j$. Also, the second failed disk must have its subscripts drawn from the set $\{i, j\}$. Likewise, it must use *both* indices, i and j, for the same reason. So reconstruction is assured unless the two failed disks are D_{ij} and D_{ji}, for some i and j. These disks are located as *reflections* of each other across the diagonal of parity disks.

In that event, the syndromes we have are

$$\begin{aligned} S_i &= D_{ji} + \alpha D_{ij} \\ S_j &= D_{ij} + \alpha D_{ji} \end{aligned}$$

From this we can derive

$$D_{ij} \;=\; (\alpha^2+1)^{-1}(S_j+\alpha S_i)$$

So we can effect reconstruction provided α^2+1 has an inverse. In GF(4), GF(8), and so forth, $\alpha^2+1 \neq 0$, so the data can be reconstructed. So the GF(4) and GF(8) ORAID disk organizations tolerate all double disk failures.

In GF(2), however, $\alpha^2+1=0$—the two syndrome equations are linearly dependent. So, in ORAID over GF(2), when two disks fail that are reflections of each other, data is lost.

The time required to reconstruct data after a failure affects reliability because it dictates the length of time that the system is exposed to additional failures. We obtain reconstruction times based upon the data bandwidths in the system. We use disk duplexing as a basis. Figure 5 shows the information flow required to reconstruct a failed disk. The failed disk is the top left one in the disk array. The figure shows the flow for two disk organization: disk duplexing and single path horizontal.

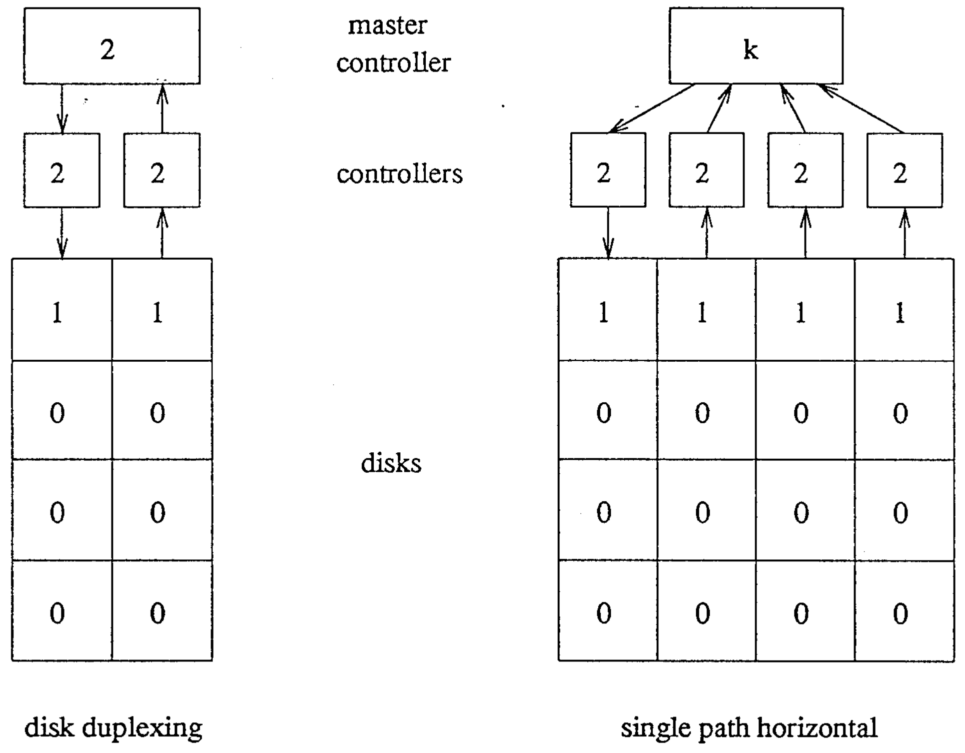


Figure 5: Load during reconstruction of single disk

The flow of information is assumed to be fully buffered. So in the left side of the figure we see that the left controller (attached to the failed disk) experiences a load of '2'. Its operations—obtaining the data from the master controller and forwarding the data to the 'failed' disk—are each counted. Buffering at each disk is ignored (this will not change between disk organizations).

The right side of the figure depicts the information flow required to reconstruct the data onto the top left disk of the disk array for the single path horizontal method. Here the master controller (or central processor) endures a notably increased load. For our purposes, we will assume that disk duplexing is a *balanced* design. By balanced, we mean that the ratio of the bandwidth demanded by reconstruction to the bandwidth available at the components is the same for each component type. So we estimate the reconstruction time after a disk failure, for the single path horizontal disk organization, to be $\max(k/2, 2/2, 1/1) = k/2$ times the reconstruction time for disk duplexing. In other words, with the single path horizontal disk organization, one can issue I/O requests for disk reconstruction at $2/k$ of the rate of disk duplexing—this induces an equivalent load at the master controller.

In practice various other considerations will come into effect. Chief among these is that the (non-reconstruction) I/O load has a pointed effect. But when comparing dramatically different disk organizations, such as disk duplexing and single path horizontal, there is a wide range of I/O load over which one (single path horizontal) is operational and the other (disk duplexing) is saturated. Using more controllers (and, to a lesser extent, using more disks) has a large impact on the I/O load that can be supported. So we regard the performance requirement of the system orthogonally. The performance requirement may veto using some disk organizations. Also, it is understood that the disk organizations using more controllers will perform better under high I/O loads.

B Reliability Analysis

We now ascertain the reliability of various disk organizations. We use the customary combinatorial approximations [3, 6, 7], except that we have made some modifications to increase accuracy. λ_d and λ_c are the rates of failure of disks and controllers, respectively. μ_d and μ_c are the rates of repair of disks and controllers, respectively. μ_c does not reflect the time needed to reconstruct out of date disks after a controller failure, because the time needed will vary with the fault circumstances. N is the number of disks in the data space. $N = k(k-1)$ for the two-dimensional disk organizations.

The expressions given are for the rates at which data unavailability occurs. The mean time to data unavailability for the I/O subsystem is equal to the reciprocal of the rate of data unavailability. First we have the disk organizations supporting single-ported disks.

Disk duplexing

$$\begin{array}{cc} & 2N\lambda_d(\lambda_d + \lambda_c)/\mu_d \\ + & 2\lambda_c(\lambda_c + N\lambda_d)/\mu_c \\ + & 2\lambda_c(\lambda_c + \lambda_d(N+1)/2)/(\mu_d/N) \end{array}$$

Here, $(N+1)/2$ arises, after a controller failure, from immunizing against each of N disks in turn.

Single path horizontal

$$\begin{array}{cc} & k^2\lambda_d(k-1)(\lambda_c + \lambda_d)/((2/k)\mu_d) \\ + & k\lambda_c(k-1)(\lambda_c + k\lambda_d)/\mu_c \\ + & k\lambda_c(k-1)(\lambda_c + \lambda_d(k+1)/2)/((2/k)(\mu_d/k)) \end{array}$$

Here, the $2/k$ factors are due to the rate at which disks can be reconstructed. The bottleneck that causes this is discussed in the previous section.

Modified EVENODD

$$
\begin{aligned}
& k(k+1)\lambda_d k\lambda_d(1/(\mu_d(2/k)))(k-1)(\lambda_c+\lambda_d/(2\mu_d(1/k)) \\
+ \quad & k(k+1)\lambda_d k(k-1)\lambda_d(1/\mu_d(2/k))(k-1)\lambda_c/(2\mu_d(2/k)) \\
+ \quad & k(k+1)\lambda_d k\lambda_c(1/\mu_d(2/k))(k-1)(\lambda_c+k\lambda_d/(\mu_d(1/k)) \\
+ \quad & (k+1)\lambda_c k^2\lambda_d(1/\mu_c)(k-1)(\lambda_c+k\lambda_d)/(\mu_d(1/k)) \\
+ \quad & (k+1)\lambda_c k\lambda_c(1/\mu_c)(k-1)(\lambda_c+k\lambda_d)/\mu_c \\
+ \quad & (k+1)\lambda_c k\lambda_c(1/\mu_c)(k-1)(\lambda_c+k\lambda_d)/((\mu_d/k)(1/k)) \\
+ \quad & (k+1)\lambda_c k^2\lambda_d(1/(\mu_d/k)(2/k))(k-1)(\lambda_c+k\lambda_d)/(\mu_d(1/k)) \\
+ \quad & (k+1)\lambda_c(k-1)\lambda_c(1/(\mu_d/k)(2/k))(k-1)(\lambda_c+k\lambda_d)/((\mu_d/(k/2))(2/k))
\end{aligned}
$$

Again, the $2/k$ factors are due to reconstruction time. In the second term, in the factor $(2\mu_d(2/k))$, the first '2' is due to disk reconstruction not being memoryless. So, after a disk failure, we first have a chance to reconstruct the disk—$(1/\mu_d(2/k))$—then after another disk failure we have a second chance to reconstruct the disk (and we assume that on average we were half way through the reconstruction).

The following are the dual-ported disk organizations.

Dual path RAID level 5

$$
\begin{aligned}
& (N+1)\lambda_d N\lambda_d/(\mu_d/((N+1)/4)) \\
+ \quad & 2\lambda_c^2/\mu_c
\end{aligned}
$$

In the first term, for the reconstruction rate $\mu_d/((N+1)/4)$, we have '4' instead of '2', because we have dual controllers. We assume that we have RAID controllers, each controller accomplishes approximately half the reconstruction work, and (since the XOR operations are done at the RAID controllers) the master controller does not experience any of the reconstruction load.

Dual path RAID level 6

$$
\begin{aligned}
& (N+2)\lambda_d(N+1)\lambda_d(1/(\mu_d/((N+1)/4)))N\lambda_d(1/(2\mu_d/((N+1)/4))) \\
+ \quad & 2\lambda_c^2/\mu_c
\end{aligned}
$$

Again, we have '4' instead of '2' in $(N+1)/4$, for the same reason as with dual path RAID level 5. Also, as we saw with the modified EVENODD disk organization, when we have a disk failure followed by another failure, we are assumed to be half way through the reconstruction of that disk.

The remaining dual-ported disk organizations follow.

Dual path horizontal

$$
\begin{aligned}
& k^2\lambda_d(k-1)\lambda_d/(\mu_d(2/k)) \\
+ \quad & k^2\lambda_d 2(k-1)\lambda_c(1/(\mu_d(2/k)))(\lambda_c+(k-1)\lambda_d)/(\mu_c+2\mu_d(2/k)) \\
+ \quad & 2k\lambda_c^2(1/\mu_c)k(k-1)\lambda_d/\mu_c \\
+ \quad & 2k\lambda_c^2(1/\mu_c)(k-1)((k+1)/2)\lambda_d/((\mu_d/k)(2/k)) \\
+ \quad & 2k\lambda_c k(k-1)\lambda_d(1/\mu_c)(\lambda_c+(k-1)\lambda_d)/(\mu_c+\mu_d(2/k))
\end{aligned}
$$

Crosshatch diagonal

$$\begin{aligned}
& k^2\lambda_d(k-1)\lambda_d/(\mu_d(2/k)) \\
+ \quad & k^2\lambda_d(k-1)\lambda_c(1/(\mu_d(2/k)))(2\lambda_c+k\lambda_d)/(\mu_c+2\mu_d(2/k)) \\
+ \quad & 2k\lambda_c k\lambda_c(1/\mu_c)(k-1)\lambda_d/\mu_c \\
+ \quad & 2k\lambda_c k\lambda_c(1/\mu_c)(k-1)\lambda_d/(\mu_d(2/k)) \\
+ \quad & 2k\lambda_c k(k-1)\lambda_d(1/\mu_c)(\lambda_c+(k-1)\lambda_d)/(\mu_c+\mu_d(2/k))
\end{aligned}$$

Crosshatch ORAID over GF(8)

$$\begin{aligned}
& k(k-1)\lambda_d^3(1/\mu_d^2)(2k-1)(9k-6)/2 \\
+ \quad & 72k^2\lambda_c^2\lambda_d^2(1/\mu_c)(1/\mu_c+(2k-1)/2\mu_d)(1/(k-1)\mu_c+(2k-1)/2\mu_d) \\
+ \quad & 32k\lambda_c^3\lambda_d(2k-1)(1/\mu_c)(1/\mu_c+(2k-1)/2\mu_d)^2 \\
+ \quad & 4k\lambda_c^4(2k-1)(1/\mu_c)(1/\mu_c+1/2\mu_d)(1/\mu_c+(2k-1)/\mu_d)
\end{aligned}$$

For the crosshatch ORAID over GF(8) disk organization, we have used GF(8) for simplicity and have broken down the failure rate into a single term for each component combination (triple disk, double disk and double controller, etc.). Except for the first term, these terms may be notably pessimistic, depending on the parameters.

C Reliability Comparison

The following plots show the mean time to data unavailability (mean time to failure, MTTF) for various disk organizations. The MTTF is calculated from the equations given in the previous section. The default parameter values for component failures are $\lambda_c = \lambda_d = 1$ per 50,000 hours. Also, the standard reconstruction rate for disk duplexing is $\mu_d = 1$ per hour. The standard controller (data path) repair rate is $\mu_c = 1$ per 4 hours; this does not include the time to reconstruct disks that have become out of date. The default number of data disks is $N = 20$ ($k = 5$).

Single-ported disks

Figure 6 shows the system MTTF for single-ported disk organizations as a function of the controller reliability. The single path horizontal disk organization is not a factor. The modified EVENODD disk organization is prevalent, except when controller reliability is quite low. With low controller reliability, the added complications of EVENODD might not be justifiable—unless the additional controller bandwidth is desired for performance reasons anyway.

Figure 7 shows the system MTTF for single-ported disk organizations as a function of the disk reconstruction time (for disk duplexing). The disk reconstruction time depends upon (among other factors) the speed and size of the disks. Faster, smaller disks favor the modified EVENODD disk organization.

Figure 8 shows the system MTTF for single-ported disk organizations as a function of the number of disks in the data space. Disk duplexing scales the best to large numbers of disks. So, for systems requiring a large number of disks and not needing the performance of the modified EVENODD disk organization, the simplicity of disk duplexing stands out.

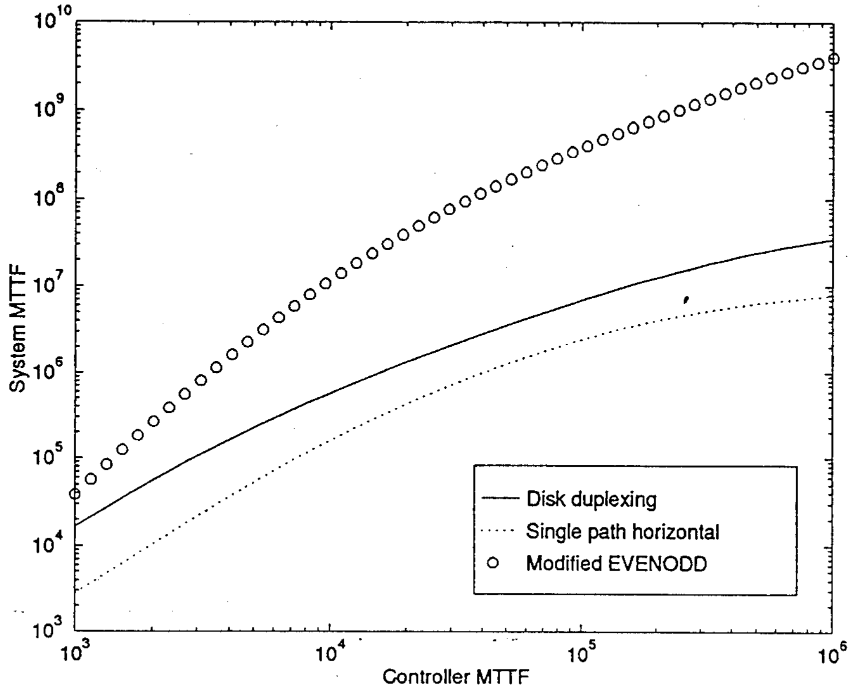


Figure 6: Longevity versus controller reliability (single-ported disks)

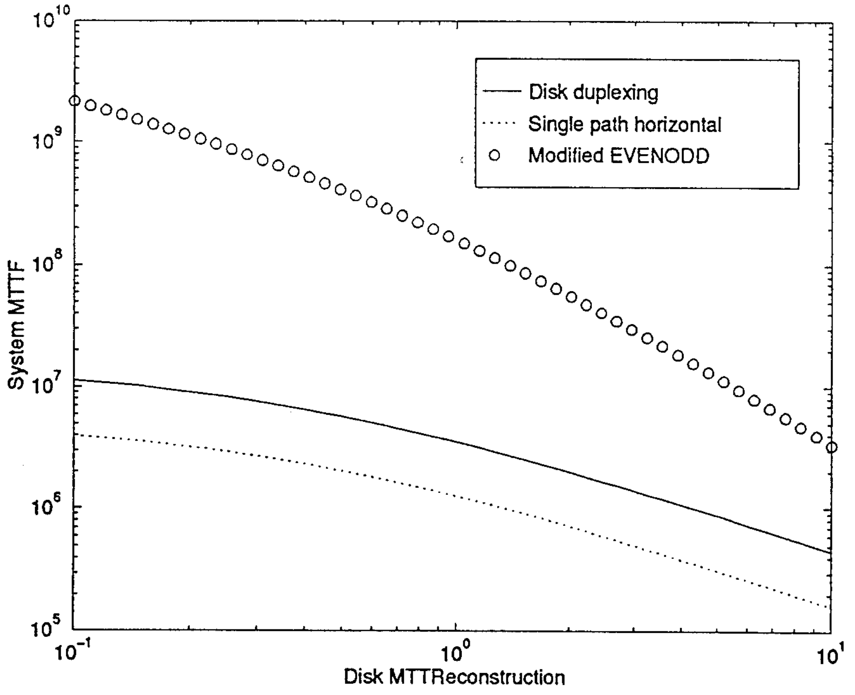


Figure 7: Longevity versus reconstruction time (single-ported disks)

This does, however, require that disk duplexing's two controllers support many disks (cf. for modified EVENODD, the high end of the plot is with 14 disks per controller).

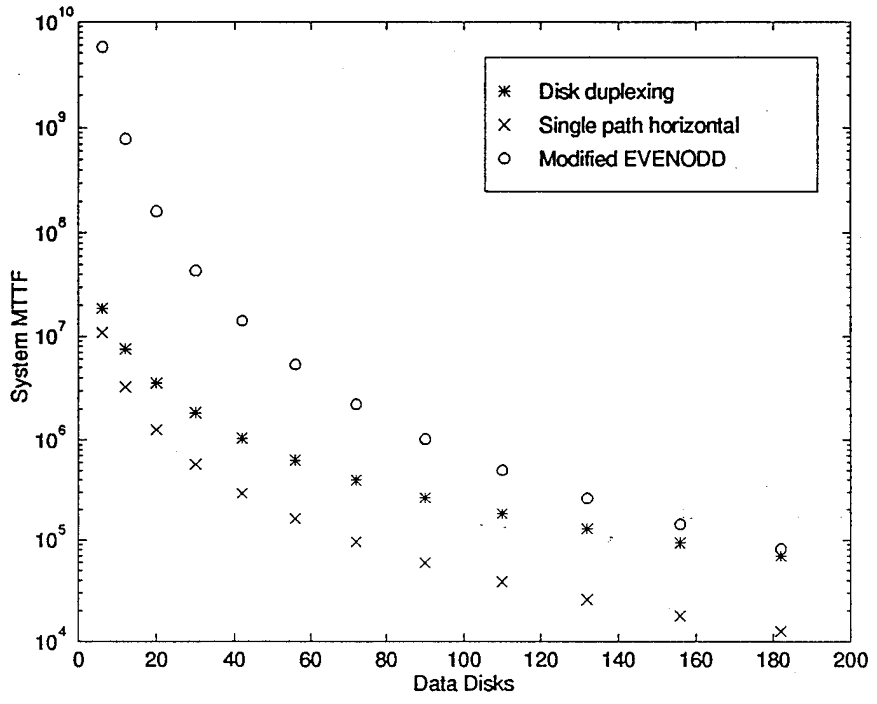


Figure 8: Longevity scalability (single-ported disks)

Greater performance can be gained using disk duplexing by increasing the number of controllers (e.g., by having 6 controllers implement 3 replicates of disk duplexing, each with a $N/3$-disk data space). This has the effect, though, of increasing system failures due to multiple controller failures. We did not investigate this.

Dual-ported disks

Figure 9 shows the system MTTF for dual-ported disk organizations as a function of the controller reliability. A surprise is that the crosshatch diagonal disk organization [7] is not distinguishing itself from the dual path horizontal disk organization [10]. We see the beginnings of a separation when controllers are very unreliable, which is the domain assumed in [7].

We also note that dual path RAID Level 6 does not stand out until controllers (data paths) are reasonably reliable. The asymptotic (in controller MTTF) reliability of dual path RAID Level 6 is notably below that of crosshatch ORAID over GF(8), because ORAID tolerates nearly all triple disk failures. The knee for ORAID also arrives at a (lower) reasonable controller reliability.

Figure 10 shows the system MTTF for dual-ported disk organizations as a function of the disk reconstruction time (for disk duplexing). This plot has no surprises; it merely confirms the superiority of crosshatch ORAID, regardless of disk size and speed.

Figure 11 shows the system MTTF for dual-ported disk organizations as a function of the number of disks in the data space. Again, this plot brings no surprises. It does make

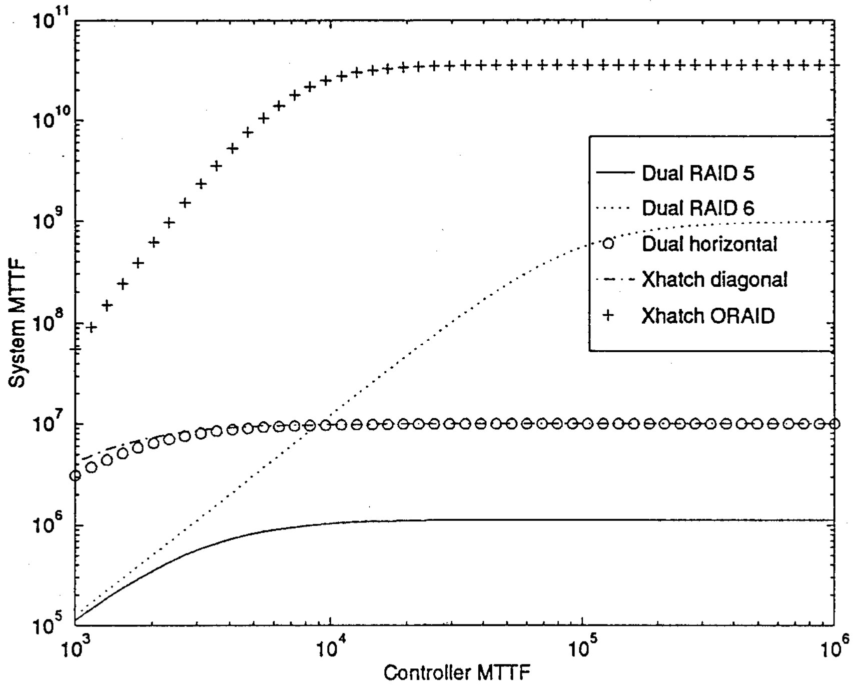


Figure 9: Longevity versus controller reliability (dual-ported disks)

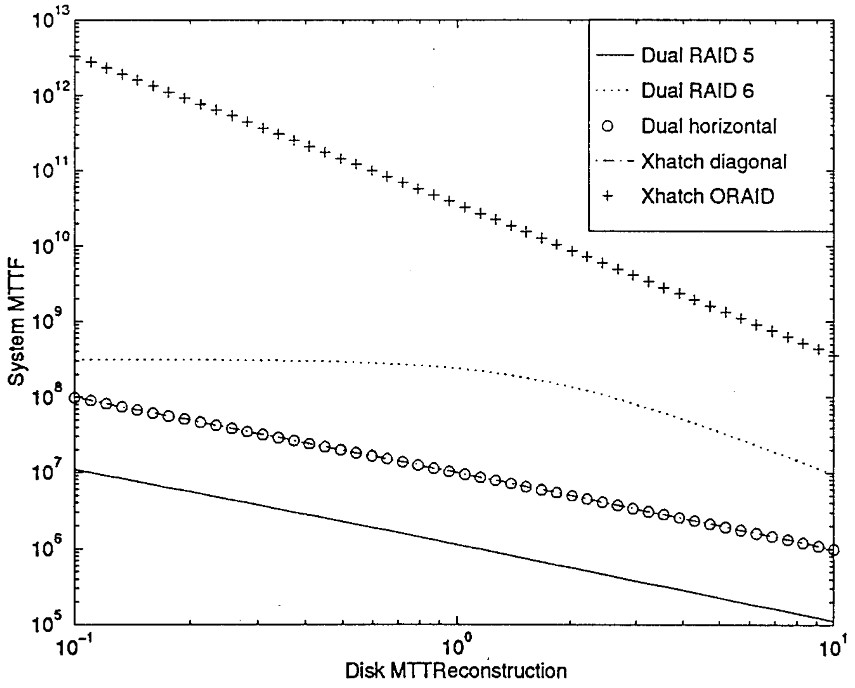


Figure 10: Longevity versus reconstruction time (dual-ported disks)

clear, however, that dual path RAID Level 6 does not scale well. For the same data space size, using fewer (larger) disks is encouraged.

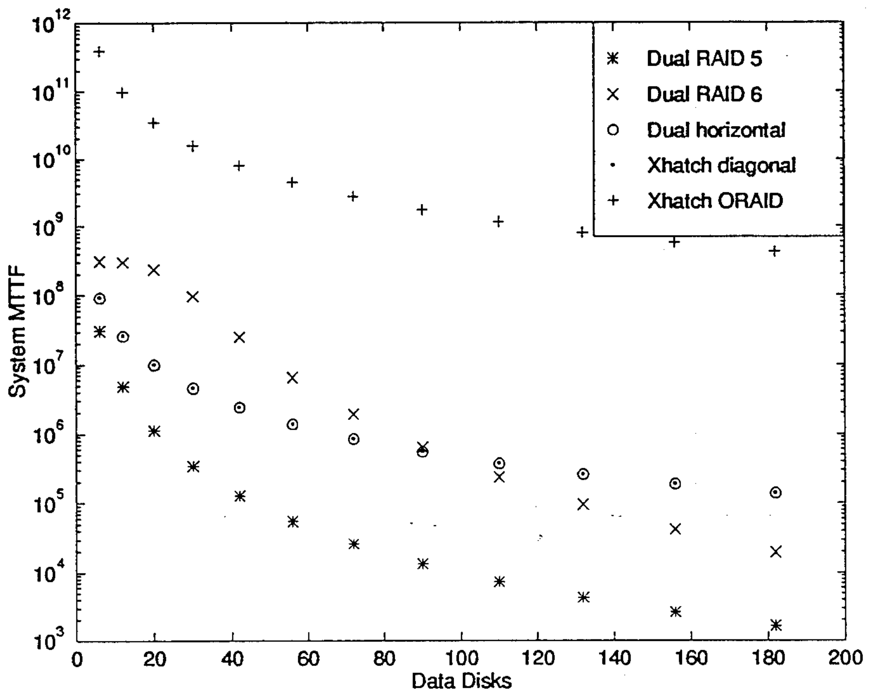


Figure 11: Longevity scalability (dual-ported disks)

II. Research in Fault Injection

Fault Injection

A Method for Validating Computer-System Dependability

Jeffrey A. Clark
*Mitre Corporation**

Dhiraj K. Pradhan
Texas A&M University

Fault injection is an effective solution to the problem of validating highly reliable computer systems. Tools such as React are facilitating its application.

With greater reliance on computers in a variety of applications, the consequences of failure and downtime have become more severe. In critical applications, such as aircraft flight control, nuclear reactor monitoring, medical life support, business transaction processing, and telecommunications switching, computing resource failures can cost lives and/or money.

Computers employed in such applications often incorporate redundancy to tolerate faults that would otherwise cause system failure. A fault-tolerant computer system's dependability must be validated to ensure that its redundancy has been correctly implemented and the system will provide the desired level of reliable service. Fault injection—the deliberate insertion of faults into an operational system to determine its response—offers an effective solution to this problem. In this article, we survey several fault-injection studies and discuss tools such as React (Reliable Architecture Characterization Tool) that facilitate its application.

COMPUTER-SYSTEM DEPENDABILITY

Dependability is a qualitative system attribute that is quantified through specific measures. The two primary measures of dependability are reliability and availability. Reliability is the probability of surviving (without failure) over an interval of time. Availability is the probability of being operational (not failed) at a given instant in time. The mean time to failure (MTTF) and the mean time between failures (MTBF) are also frequently used. Dependability is often evaluated empirically through life testing. However, the time needed to obtain a statistically significant number of failures makes life testing impractical for most fault-tolerant computers. Instead, analytical modeling is typically used to predict dependability.

Analytical dependability models enumerate a system's operational or failed states. Each state represents a unique combination of faults and their effects on system components. The times at which the faults occur are assumed to fit a particular statistical distribution. Several standardized procedures estimate the failure rates of electronic components when the underlying distribution is exponential. However, fault handling beyond this stage has been modeled in many different ways.

Most fault-handling models use coverage parameters to specify the probability of successfully performing the actions needed to recover from a fault. These actions include detecting the fault, identifying the affected component, and isolating that component through system reconfiguration. Each action must be taken quickly, before any additional faults that can overload the system's fault-handling mechanisms accumulate. For this reason, many models incorporate distributions of latency—the time needed to perform each of these actions. Because even small variations in coverage and latency can greatly affect dependability, these parameters should be estimated based on data from the actual system rather than approximated (see "Background" sidebar).

Fault-injection studies can provide this data through many individual experiments that vary how, where, and when the faults are intentionally

* *The views, opinions, and/or findings in this article are those of the authors, and should not be construed as the official positions, policies, and/or decisions of the Mitre Corporation or its government sponsors.*

0018-9162/95/$4.00 © 1995 IEEE

inserted. Large complex systems and time constraints make exhaustive insertion impractical; therefore, only a carefully chosen subset of all possible faults can usually be investigated. Insertion must be controlled so that the type, location, time, and duration of each fault, or the corresponding statistical distributions, are at least approximately known. Faults can be inserted into both the hardware and software components of a realized system or a simulation model that accurately reflects these components' behavior. During each experiment, the system must be operated with a representative work load to obtain a realistic response. The effects of each inserted fault are precisely monitored and recorded with instrumentation.

Besides supplying coverage and latency parameters for analytical models, fault injection can directly evaluate dependability metrics. It is particularly useful for measuring those system attributes that are difficult to model analytically—for example, the work load's influence on dependability. Fault injection aids design when it is used to functionally test a prototype during system development. It can identify implementation errors in fault-tolerance mechanisms and provide feedback on those mechanisms' efficiency. When the system is ready for deployment, fault injection can be used to observe the error or failure symptoms associated with each type of faulty component. Fault dictionaries can then be compiled to support system diagnosis during maintenance actions. Finally, fault-injection experiments provide a means for understanding how computer systems behave in the presence of faults. Such knowledge will ultimately lead to better system designs and higher dependability.

TAXONOMY OF EXPERIMENTS

Fault-injection experiments can be classified according to three general attributes: system abstraction, fault model and injection method, and dependability measure.

System abstractions

Fault-injection studies have traditionally been performed on the actual hardware and software of physical

Background

A *fault* is a deviation in a hardware or software component from its intended function. Faults can arise during all stages in a computer system's evolution—specification, design, development, manufacturing, assembly, and installation—and throughout its operational life. Most faults that occur before full system deployment are discovered through testing and eliminated. Faults that are not removed can reduce a system's dependability when it is in the field. Despite the potential for such latent faults in computer systems, most fault-injection studies focus on the faults that occur during system operation.

Hardware faults occurring during system operation are categorized mainly by duration. *Permanent faults* are caused by irreversible device failures within a component due to damage, fatigue, or improper manufacturing. Once a permanent fault has occurred, the faulty component can be restored only by replacement or, if possible, repair. *Transient faults*, on the other hand, are triggered by environmental disturbances such as voltage fluctuations, electromagnetic interference, or radiation. These events typically have a short duration, returning the affected circuitry to a normal operating state without causing any lasting damage (although the system state may continue to be erroneous). Transients can be up to 100 times more frequent than permanents, depending on the system's particular operating environment. *Intermittent faults*, which tend to oscillate between periods of erroneous activity and dormancy, may also surface during system operation. They are often attributed to design errors that result in marginal or unstable hardware.

Software faults are caused by the incorrect specification, design, or coding of a program. Although software does not physically "break" after being installed in a computer system, latent faults or bugs in the code can surface during operation—especially under heavy or unusual work loads—and eventually lead to system failures. For this reason, software fault injection is employed primarily for testing programs or software-implemented fault-tolerance mechanisms. However, it has not seen widespread use in either application.

When a fault causes an incorrect change in machine state, an *error* occurs. The time between fault occurrence and the first appearance of an error is called the fault latency. Although a fault remains localized in the affected code or circuitry, multiple errors can originate from one fault site and propagate throughout the system. If the necessary mechanisms are present, they will detect a propagating error after a period of time, called the error latency. When the fault-tolerance mechanisms detect an error, they may initiate several actions to handle the fault and contain its errors. *Recovery* occurs if these actions are successful; otherwise, the system eventually malfunctions and a *failure* occurs.

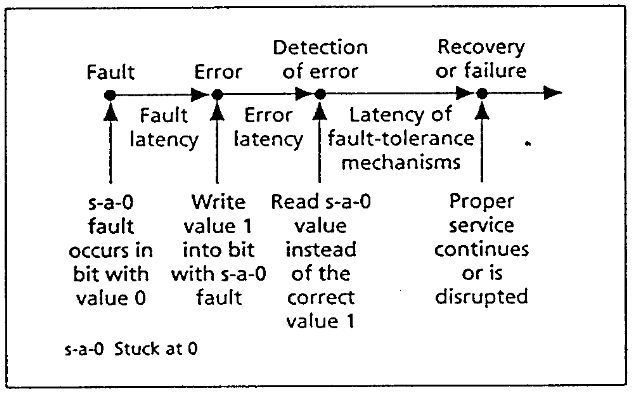

Fault
Error
Detection of error
Recovery or failure
Fault latency
Error latency
Latency of fault-tolerance mechanisms
s-a-0 fault occurs in bit with value 0
Write value 1 into bit with s-a-0 fault
Read s-a-0 value instead of the correct value 1
Proper service continues or is disrupted
s-a-0 Stuck at 0

Figure A. Example of a fault, an error, and a failure.

Figure A provides an example to clarify the definitions of fault, error, and failure. Suppose a permanent stuck-at-0 (s-a-0) fault affects a memory bit with an initial value of logical 0. Some time later, an error occurs when a logical 1 is written into this bit. (If the faulty value had been opposite the initial value of this bit, an error would have manifested immediately with no fault latency.) The next read from the memory bit obtains the s-a-0 value instead of the correct value, 1, thereby detecting an error. Proper service continues if the system's fault-tolerance mechanisms can correct or mask this bit error. If not, service is disrupted.

computer systems. High levels of device integration, multiple-chip hybrid circuits, and dense packaging technologies limit accessibility to injection and instrumentation nodes. This makes it difficult to validate the hardware of physical systems. Simulation, on the other hand, has the advantage of relatively uninhibited access to a modeled system's internal nodes. The ability to precisely control and monitor injected faults, coupled with low-cost computer automation, and the potential for earlier application make simulated injection an attractive alternative to physical injection.

Simulated fault injection can support all system abstraction levels—architectural, functional, logical, and electrical. Mixed-mode simulation, where the system is hierarchically decomposed for simulation at different abstraction levels, is particularly useful for fault injection. This technique lets faults be accurately simulated at a low abstraction level, while the system responses are efficiently simulated at higher abstraction levels.

Fault models and injection methods

Simulated fault injection and most experiments involving physical hardware and software require selection of a fault model. The popular stuck-at fault model is commonly used for permanent hardware faults. However, subsequent errors often are of more concern than the faults themselves. This is particularly true for transient faults, whose unpredictable origin and relatively short life span make them difficult to characterize. Therefore, studies involving transients frequently employ an inversion model, where a fault immediately produces an error with the opposite logical value. Software errors arising from hardware faults are often modeled via bytes of 0s or 1s written into a data structure or portion of memory. Experimenters can use various other models, from detailed device-level to simplified functional-level models, to represent faults or their manifestations.

After choosing a fault model, the experimenter must determine how to inject the faults into the computer system. Locations frequently exploited when faults are injected into physical systems include IC leads, circuit board connectors, and the system back plane. The experimenter can generate faults at these external sites by temporarily inserting circuitry that corrupts the signals passing through a node without damaging any system components. Although signal corruption can model many faults that occur inside components, this method usually does not exercise all relevant hardware in the system. Therefore, experimenters cannot investigate the effects of some internal faults with this injection technique.

State mutation is one method of injecting errors inside system components. During normal system operation, processing is halted and special-purpose hardware or software is used to introduce errors. Scan paths, designed for system test and diagnosis, can be used to read the shift-register contents, modify selected bits, and shift the mutated state back into the machine. Privileged system calls and program debuggers can insert errors into a computer system by directly modifying its memory or register state. State mutation is the injection method used most often with simulated fault injection. Computer simulators are typically event driven, updating a modeled system's state at discrete times rather than continuously. Fault injections are easily made between event time boundaries. However, because it requires stopping and restarting the processor to inject a fault, this technique is not always effective for measuring latencies in physical systems.

Several novel approaches exist for injecting internal faults in hardware. ICs are susceptible to single-event upsets (SEUs)—created when an ionizing particle passes through a transistor, generating excess charge. Computer systems in space applications are particularly vulnerable to SEUs from cosmic rays. In the laboratory, transient faults can be induced in a similar way through short-term exposure to heavy-ion radiation. However, these fault-injection experiments must be performed in a vacuum chamber with the lid of the target IC removed, since ions are easily attenuated by air. Radiation flux is distributed uniformly over the chip, and error rates can be adjusted by a change in the distance from the ion source. Shielding can confine faults to a particular region of the IC, but there is no direct control over where and when the injections occur.

Another means for injecting internal hardware faults is through power supply disturbances. Short, pulsed interruptions in power drop the supply voltage to levels that can increase propagation delays and discharge nodes, especially those in memory. Computer systems employed in industrial applications are often subject to similar noise on the power lines. Unlike radiation, which causes SEUs, power supply disturbances simultaneously affect many nodes in the target IC, producing multiple, transient bit faults. Unfortunately, the location of these faults cannot be readily controlled. This injection technique is quite sensitive to the pulse width and amplitude of the voltage disturbances. Effects can also vary widely with different circuit families and fabrication technologies, making it difficult to generalize results from such experiments.

The last method we consider for introducing faults into a computer system is called trace injection. This method first uses custom-monitoring hardware or software to periodically sample machine state or record memory references on an operational system. Then the acquired trace is used to simulate system behavior, as errors that mimic faults in the instrumented components are inserted into the trace. The quantity of data collected can be very large, limiting most traces to only a brief history of machine activity. It is therefore essential to associate some measure of system load (at the time the trace was obtained) with the results, to distinguish extremes in fault behavior from the norm.

Dependability measures

The traditional objective of fault-injection experimentation has been to estimate coverage and latency parameters for analytical dependability models. However, fault injection can also evaluate other dependability measures, including reliability or availability and MTTF or MTBF. Several failure classification experiments have analyzed how injected faults affect a computer system's service. Fault-injection studies have also investigated error propagation from a fault site to other system components. Finally, researchers have often observed a correlation between a system's dependability and either its computational load or characteristics of its application code. Such work load relationships are frequently explored via fault injection. Figure 1 summarizes the system abstractions, injection methods, and dependability measures for classifying fault-injection experiments.

APPLICATIONS

Fault injection was first employed in the 1970s to assess the dependability of fault-tolerant computers. For some time afterward, fault injection was used almost exclusively by industry for measuring the coverage and latency parameters of highly reliable systems. Not until the mid-1980s did academia begin actively using fault injection to conduct experimental research. Initial work concentrated on understanding error propagation and analyzing the efficiency of new fault-detection mechanisms. Research has since expanded to include characterization of dependability at the system level and its relationship to work load.

Error propagation in a jet-engine controller

We first examine a study that explored error propagation in an HS1602 jet-engine controller with dual-channel redundancy. Choi and Iyer used the Focus simulation environment to inject transient faults into one of the two microprocessors in this controller.[1] They used mixed-mode simulation at the electrical and logical levels to deposit 0.5 to 9 pico-coulombs of excess charge onto different nodes of the microprocessor as it executed a phase of its application code. The excess charge models transients from the penetration of various heavy ions typically found in cosmic environments. The data in Table 1 is from a comparison of 2,100 simulated fault-injection experiments with a trace of the fault-free simulation. First-order errors are those manifested in the first clock cycle after fault injection. Errors manifested in the second and subsequent clock cycles are called second- and higher-order errors, respectively. Results indicate that nearly 80 percent of the injected transients had no impact, since errors had to be latched (stored in a memory element) to affect the microprocessor's state. Once latched, however, an error had more than a 50 percent chance of reaching a pin and more than a 40 percent chance of causing a functional error on the microprocessor's control outputs. By analyzing the individual contributions to these statistics by each of the HS1602's six functional units, Choi and Iyer discovered the most effective locations for incorporating additional fault-tolerant features.

Table 1. Transient fault severity.

Type	Percentage
First-order latch errors	22.4
Second- and higher-order latch errors	5.7
First-order pin errors	2.1
Second and higher-order pin errors	4.3
Functional errors	9.2

Radiation and power supply disturbances

Karlsson et al. used radiation and power supply disturbances to investigate the propagation of internal errors to the bus of an MC6809E.[2] They injected transient faults into this microprocessor by exposing it to heavy ions from a Californium source and to – 4.2 V, 50-ns pulses on the microprocessor's 5V power supply. A reference MC6809E ran the same two test programs in lock-step synchronization with the microprocessor under test. Comparison of bus signals from the two microprocessors detected errors. Detection triggered a logic analyzer to record microprocessor activity for 200 bus cycles. Table 2 lists the bus affected in the first erroneous cycle based on 1,000 observations. Errors appeared mainly on the address bus in the radiation experiments, whereas errors on the control bus dominated the power-supply disturbances. Although the initial fault manifestations were quite different, the microprocessor's behavior over an extended period of time was almost identical for both injection techniques. As Table 3 shows, control-flow errors causing permanent divergence

Table 2. Bus affected in the first erroneous cycle.

Bus affected	Heavy-ion radiation (percent)	Power supply disturbances (percent)
Address	64	17
Data	5	1
Control	27	80
Combination	4	2

System abstractions	
Physical	Logical
Architectural	Electrical
Functional	Mixed-mode
Injection methods	**Dependability measures**
Signal corruption	Reliability/availability
State mutation	MTTF/MTBF
Radiation	Failure classification
Power supply disturbances	Coverage and latency
Trace injection	Error propagation
	Work load relationships

Figure 1. Summary of the experimental taxonomy.

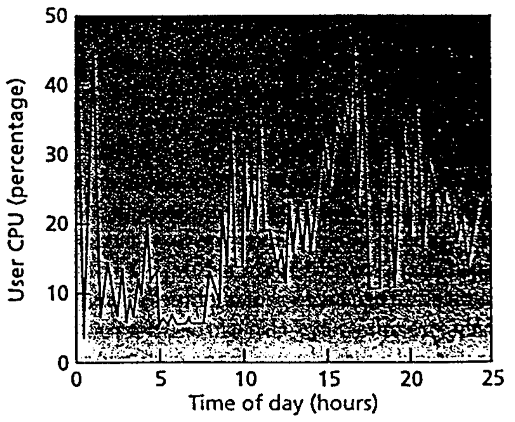


Figure 2. User CPU usage by time of day.

from the correct instruction stream were responsible for over 70 percent of the failures observed with heavy-ion radiation and power-supply disturbances. Karlsson et al. have evaluated the coverage and latency of several different concurrent error-detection schemes using these methods.

Trace injection to measure latency

Chillarege and Iyer were among the first to measure fault and error latency in memory via trace injection.[3] They ran a scanning process on a VAX 11/780 to periodically copy the contents of real memory locations into archival storage. The locations were randomly chosen from 4 to 10 regions in memory of up to 50 Kbytes each. These regions were repetitively scanned every 15 to 20 seconds under a medium to high system work load. Stuck-at bit faults were then simulated in the sampled words to calculate latency distribution parameters (given in Table 4) for a representative set of 960 faults. The mean fault latency was almost five times greater for s-a-0 (stuck-at-0) than for s-a-1 faults. Conversely, the mean error latency of the s-a-1 faults was more than double that of the s-a-0 faults. Chillarege and Iyer attributed the difference in latencies to unequal lifetimes of 0s and 1s in the system due to the way memory is allocated and released. They conjectured that many programs use only a fraction of their allocated memory blocks. This would leave many 0s in memory, because blocks are initially cleared when they are allocated. Optimal memory scrubbing rates—the frequency at which single, transient bit errors are systematically corrected before any additional errors accumulate—are determined from such measurements of fault and error latency.

System work load and memory error latency

Chillarege and Iyer also used trace injection to analyze the relationship between system work load and memory error latency.[4] They collected data by probing the back plane of a VAX 11/780 and sampling physical memory activity at 40-second intervals. They also logged work load profiles during this data acquisition. Figure 2 graphs one measure of system work load, user CPU utilization (percentage of processing capacity in use), over a 24-hour period beginning at midnight. Work load was relatively low until shortly after 7 a.m. (except for a brief period around 1 a.m., when system routines were run), then rose significantly between 8 and 10 a.m., and peaked in the mid- to late-afternoon. Chillarege and Iyer used the memory activity data to simulate inverted bit errors occurring at different times of day. Error latency distributions for faults inserted at midnight and noon appear in Figures 3 and 4, respectively. Mean error latency varied from as long as eight hours at low work load to as short as 44 minutes

Table 3. Classification of processor errors.

Error class	Heavy-ion radiation (percent)	Power supply disturbances (percent)
Control-flow errors		
Permanent divergence	72	74
Temporary divergence	3	4
Not active within 200 cycles	2	0
Data errors		
Data only	5	2
Address/control also affected	15	16
Other errors		
Could cause failure	4	2
Could not cause failure	0	3

Table 4. Memory latency distribution parameters. All latencies are in minutes.

	Stuck at 0		Stuck at 1	
Latency	Mean	Standard deviation	Mean	Standard deviation
Fault	70.4	80.2	14.6	31.9
Error	20.6	31.2	45.4	47.9
Total	91.1	76.9	60.4	47.6

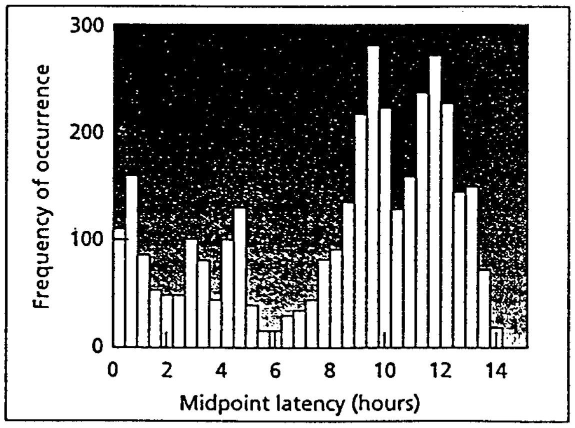


Figure 3. Error latency distribution for a fault at midnight.

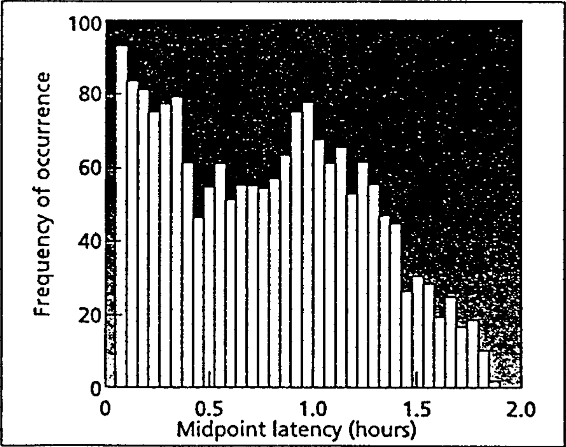


Figure 4. Error latency distribution for a fault at noon.

Table 5. Completion category distributions.

Completion category	Matrix multiplication (percent)	Recursive Fibonacci computation (percent)
Overwritten	64	71
Fatal errors	17	8
Time-outs	7	7
Results wrong	8	8
Results OK	4	6

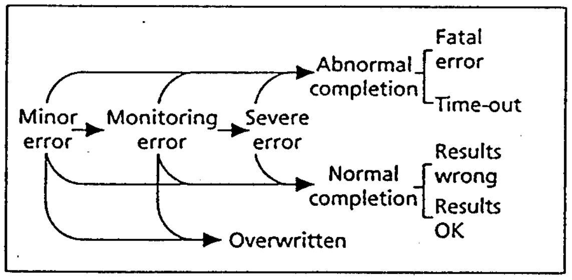


Figure 5. Fault manifestation and error propagation.

at high work load. Notice that a fault occurring at midnight was likely to remain dormant until the sharp increase in work load beginning at 8 a.m., whereas a fault at noon had a high probability of being detected quickly. This clearly demonstrated that error latency strongly depends on the work load following the fault's occurrence.

Impacts of faults on program behavior

Czeck and Siewiorek employed simulated fault injection to study the effects of gate-level faults on program behavior in the IBM RT PC.[5] They exhaustively injected one-cycle inversion faults into 10 key CPU locations across the entire execution time of a matrix multiplication and a recursive Fibonacci program. They incorporated several different error-detection mechanisms (EDMs) into this processor's simulation model. Figure 5 illustrates possible fault manifestations and error propagation to the EDMs. An injected fault initially caused a minor error. If the minor error later propagated to and was detected by an EDM, it became a monitoring error. A severe error occurred when a monitoring error disrupted control flow. The program would then either complete with correct or incorrect results or terminate through a time-out or fatal error. Table 5 reports the outcomes for both work loads. Of the 18,900 transients injected, 60 to 70 percent were inserted into idle hardware in the processor and eventually overwritten. Of those faults that were not overwritten, approximately 30 to 40 percent lead to normal program completion, while over 60 percent produced severe errors. Czeck and Siewiorek later developed a model predicting faulty system behavior from work load attributes such as instruction type, control flow structure, and instruction mix, based on these experimental results.

Failure acceleration

Chillarege and Bowen introduced the concept of failure acceleration to increase the speed at which a system transitions between the good, erroneous, and failed states during fault-injection experiments.[6] They accomplished this by decreasing fault and error latency and increasing the probability of a fault causing a failure, without altering the fault model. The idea was utilized in a study involving 70 experimental runs that filled a random page of real storage in an IBM 3081 mainframe with bytes of hexadecimal FF. This faulty bit pattern emulates the effects of a software overlay, which arises when a program writes into an incorrect storage area. During the experiment, the system executed simulated on-line database transactions that kept CPU utilization between 85 and 90 percent. The resulting state transition diagram (depicted in Figure 6) indicates that only 16 percent of the injected faults caused the system to quickly crash. One third of the observations were classified as partial failures, representing some loss in service without any adverse effect on the primary application. In 51 percent of the experimental runs, nothing happened within 15 minutes of the fault injection. Roughly half of these responses were later identified as potential hazards, or errors that had caused significant damage to the system but—under the prevailing operating state—would remain dormant. There was adequate time to repair 60 percent of the errors that did not affect

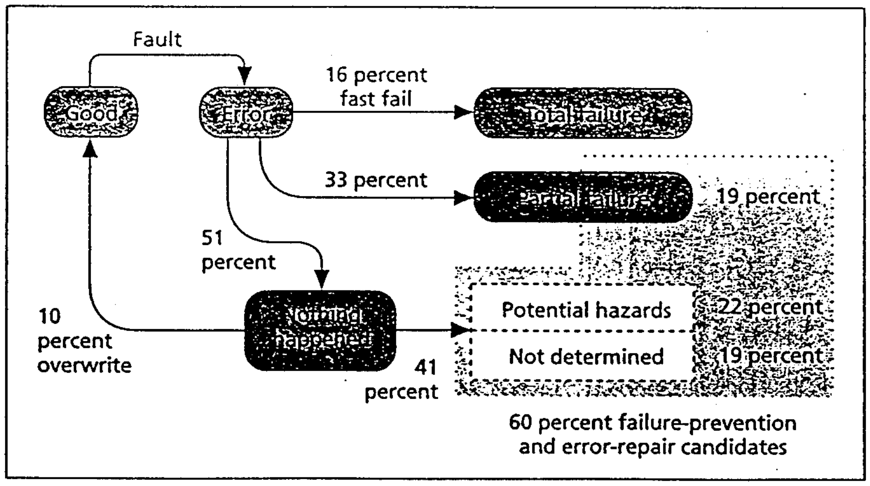


Figure 6. State transitions under failure acceleration. All faults produce errors. 16 percent cause failure quickly, and 33 percent cause a partial failure—with 19 percent being partial failures that are failure-prevention or error-repair candidates. For 51 percent of faults, nothing happens—with 10 percent being overwritten and 41 percent remaining as potential hazards or "not determined." 60 percent of all faults are failure-prevention or error-repair candidates.

the short-term availability of the system. Chillarege and Bowen discussed failure prevention and error repair techniques to detect and remove these errors and avert the loss of primary service.

Transient errors impact availability

Goswami and Iyer explored the impact of latent and correlated transient errors on a commercial fault-tolerant system's availability.[7] The target for this study was the triple-modular redundant (TMR) processing core of the Tandem Integrity S2. Processor modules are triplicated in this machine, and a majority voter masks erroneous outputs from any one processor. Goswami and Iyer used the Depend tool to inject transients into a functional-level simulation of the system's CPUs and memories. They simulated system operation 10 to 60 times, over periods of up to 200 years, to obtain statistically significant MTBF estimates. They considered three different error arrival rates (λ_1 = 1/24 hours, λ_2 = 1/72 hours, and λ_3 = 1/120 hours) and latencies, based on the analysis of real error data collected from other systems. The results graphed in Figure 7 show that latent transients alone did not adversely affect the system's MTBF. However, when 85 percent of the injected errors were correlated by even a small percentage, the degradation in MTBF was enormous. To sustain a high MTBF in the presence of latent errors, Goswami and Iyer suggested frequent memory scrubbing and reducing the time required for a CPU power-on self-test. In other experiments, they measured the coverage and latency of two memory-scrubbing schemes running under a simulated application program.

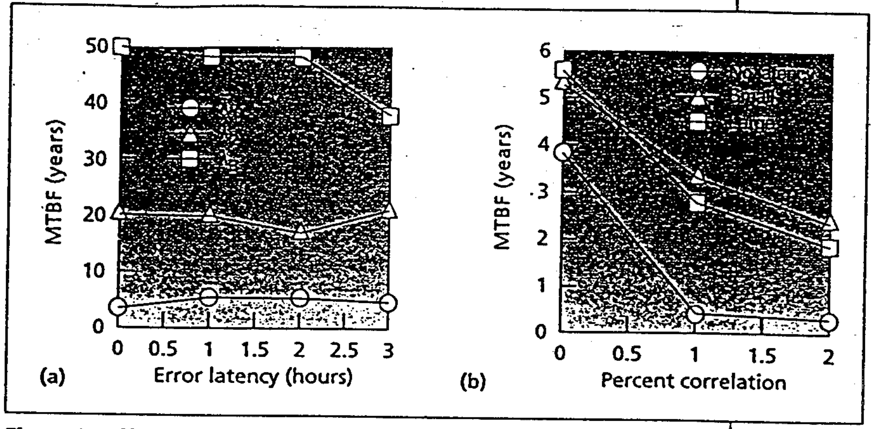


Figure 7. Effect of latent and correlated errors on MTBF (mean time between failures): (a) uncorrelated latent errors; (b) correlated latent errors.

Evaluating proposed designs

The studies discussed so far focused on validating existing systems, but fault injection can also evaluate the dependability of proposed designs. We have used simulated fault injection to analyze the reliability of several alternative TMR architectures.[8] Bidirectional voting (BDV) on both memory read and write accesses is typically performed in TMR systems. We proposed read-only voting (ROV) and write-only voting (WOV) to reduce the voting performance penalty through a small sacrifice in reliability. We used the React (Reliable Architecture Character-ization Tool) fault-injection testbed to empirically compare these three different designs. React simulated each TMR system's processors, memories, and voter at the functional level. The processors executed a synthetic work load, while permanent and transient faults were injected into the system components at exponentially distributed interarrival times. Figure 8 shows the reliability/performance tradeoff obtained via unidirectional voting. One million TMR systems of each type were simulated over a 100-hour mission to generate these plots. For equal processor and memory module failure rates (λ_P and λ_M, respectively) in the upper plot, the reliability was significantly higher for BDV than for either the ROV or WOV architecture. However, when the memory failure rate was 10 times greater than the processor failure rate, the difference between the reliability curves shrank in the lower plot. Our results indicate that in many cases, the unidirectional-voting TMR systems give up a little reliability for a potentially large increase in performance.

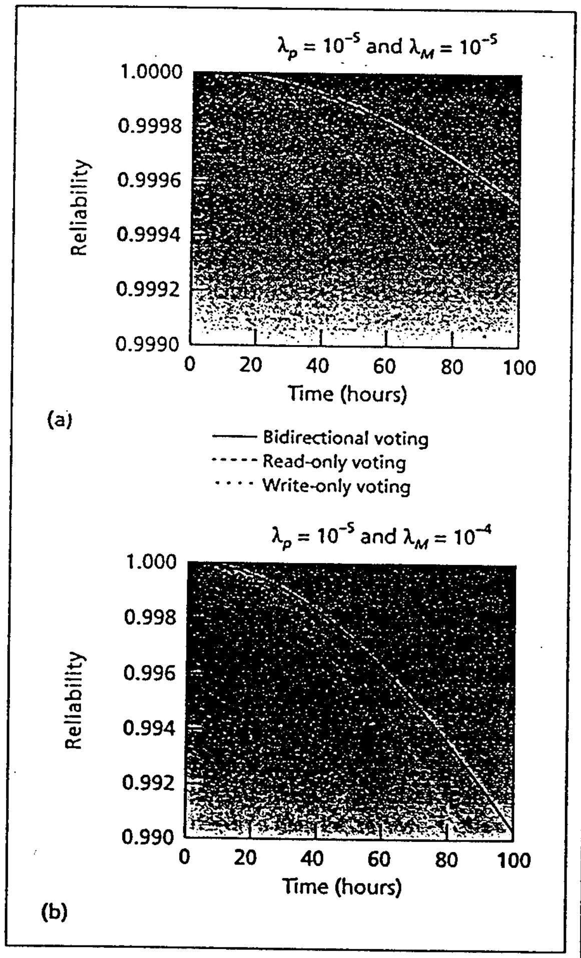


Figure 8. Reliability trade-off of the alternative triple-modular redundant (TMR) designs: (a) equal processor and memory module failure rates ($\lambda_P = \lambda_M = 10^{-5}$ failures/hour); (b) memory module failure rate greater than processor module failure rate.

FAULT-INJECTION TOOLS

Most fault-injection experiments were not designed around a formalized methodology. Experimenters typically developed customized approaches to validate each new system. This makes it difficult to apply specific results from different studies when analyzing other systems. Moreover, the complexity of today's systems can make the fault space (defined as fault type × location × injection time) huge. This means many experiments must be performed to achieve statistical confidence in the dependability metric being measured. To obtain the most accurate results in the shortest time, we must accelerate the injection and measurement processes. Fault-injection tools address these problems by integrating models, methods, and measurements into a generalized framework for conducting automated experiments on a variety of systems.

Messaline

Various fault-injection tools can evaluate physical systems, but few offer the versatility of Messaline, which was developed by LAAS-CNRS (Laboratory for the Analysis of System Architectures at the National Center for Scientific Research), France.[9] Its design is based on a formalized fault-injection methodology. The result is a flexible testbed capable of simultaneously injecting multiple, pin-level faults into different target systems to collect coverage, latency, and error-propagation measurements. A host computer manages fault injection by generating the test sequence, providing runtime execution control, and archiving data for analysis. Messaline has validated a centralized computer interlocking system for railway control and the distributed communication system of the Esprit Delta-4 project.

Fiat

The Fault-Injection-Based Automated Testing environment combines the flexibility of software control with hardware emulation, to evaluate the dependability of fault-tolerant distributed systems.[10] Fiat uses software-implemented fault injection to (erroneously) set and clear bytes in the memory images of programs. The programs execute on a network of machines configured to model a particular system architecture. This tool was realized with four IBM RT PCs connected via a token ring at Carnegie Mellon University. Fiat has been used to measure coverage and latency, classify failures, and investigate the effects of fault type and work load on these metrics.

Ferrari

The Fault and Error Automatic Real-Time Injector was designed at the University of Texas to estimate the coverage and latency of fault-tolerance mechanisms.[11] Like Fiat, it uses software-implemented injection to emulate hardware faults. However, instead of injecting errors directly into memory, Ferrari traps instructions affected by the fault so that a routine can be executed to mimic system behavior in the presence of the real fault. Various permanent and transient hardware faults, program control-flow errors, and user-defined faults/errors can be injected. Running on a Sun SparcStation under X Windows, Ferrari has evaluated the effectiveness of several concurrent error-detection techniques embedded in application software.

Focus

The Focus simulation environment conducts fault sensitivity experiments on chip-level designs.[1] Transient faults are injected through a runtime modification of the circuit, whereby a time-dependent current source is added to a device-level node. The current source deposits excess charge on this node to represent the penetration of an alpha particle or other electrical disturbance. The software provides various statistical measures to quantify fault sensitivity, including charge thresholds, error distributions, and two state-transition models that describe error generation and propagation. Focus uses a graphical analysis facility for Sun workstations, letting it visualize fault activity in a chip's functional units and error propagation on the major interconnects to external pins. Focus was developed at the University of Illinois and was used to analyze a dual-channel jet-engine controller.

Depend

The Depend environment is a joint dependability and performability evaluation tool that analyzes fault-tolerant architectures at the system level.[7] This process-based simulator provides a library of objects to behaviorally model a system's hardware components. Using these objects, a control program written in C++ simulates system operation and models system software. The objects automatically inject faults, initiate repairs, and compile statistics—such as the number of failures per component and the component's MTBF—that can be graphically displayed or included in a report. Permanent, transient, and user-defined faults can be injected with latency or at correlated times. A fault-injection scheme based on work load is also available. Depend was developed at the University of Illinois and has been used to analyze the Tandem Integrity S2 commercial fault-tolerant processor and a load-sharing distributed system.

React is a software testbed that abstracts multiprocessor systems at the architectural level.

React

In a cooperative effort between the University of Massachusetts and Texas A&M, our group has produced the Reliable Architecture Characterization Tool.[12] React is a software testbed that abstracts multiprocessor systems at the architectural level. It performs life testing through simulated fault injection to measure dependability. This involves conducting a statistically significant number of experiments or trials, each simulating the operation of an initially fault-free system. Randomly occurring faults are injected into each system until it fails or reaches a specified censoring time. Failure statistics are collected during each trial and are later aggregated over the entire simulation run to compute dependability metrics.

We have incorporated detailed system, work load, and fault/error models into the React software. Figure 9 depicts the system model employed by React. This class of architectures contains one or more processor modules (P) interconnected via buses (B) to one or more memory modules (M) through a block of fault-tolerance mechanisms. The fault-tolerance mechanisms supply the hardware necessary to detect, correct, or mask errors during memory accesses and to reconfigure the system when modules fail. This framework provides the flexibility needed to represent many different architectures without requiring custom simulation models for each one. React can analyze multiprocessor systems that use *N*-modular redundancy, duplication and comparison, standby sparing, or error-control coding to achieve fault tolerance.

React assumes a synthetic work load. Processors continually perform instruction cycles consisting of several possible memory references and the simulated execution of an instruction. React does not use real application code and data, but allows errors to propagate throughout the system as if the software were actually being executed. The work load model is specified by a mean instruction execution rate, the probabilities of performing a memory read and write access per instruction, and a locality-of-reference model that determines which locations are accessed. These parameters can easily be extracted from memory reference traces collected during application software development.

Permanent and transient faults can be automatically injected into a system's processors, memories, and fault-tolerance mechanisms. Fault occurrence times are sampled from a Weibull distribution. Faults affect a processor's data and control paths and a memory's bit-array and addressing logic. Each faulty component's erroneous behavior is governed by a stochastic model that accounts for both fault and error latency. We derived these stochastic models from the results of other low-level fault-injection experiments. Repair times for failed components are assumed to have a lognormal distribution after a fixed logistics delay. The time required to reintegrate a repaired component back into the system and the time to reboot the system after a critical failure are constant and user specified.

We demonstrated the effectiveness of React by analyzing several alternative multiprocessor architectures. Specifically, we investigated two dependability tradeoffs associated with triple-modular redundant (TMR) systems. The first study explored the reliability/performance tradeoff in voting unidirectionally instead of bidirectionally on either memory read or write accesses. The second study examined the reliability/cost tradeoff in duplicating and comparing (via error-detecting codes) the memory modules rather than triplicating and voting on those modules. Both studies showed that a small sacrifice in reliability can be made for potentially large performance increases or cost reductions compared to traditional TMR design.

FAULT INJECTION HAS BECOME A VALUABLE ASSET for evaluating computer system dependability. It has been used to obtain analytical-model parameters, validate existing fault-tolerant systems, and synthesize more reliable system designs. However, many problems remain.

One challenge is to reduce the large fault space associated with highly integrated systems. This will require improved sampling techniques and models that equivalently represent the effects of low-level faults at higher abstraction levels. The impact of specification and design faults, particularly in software, is another largely unexplored problem. A better understanding of their occurrence is necessary before we consider injecting specification and design faults to validate computer-system dependability. Another obstacle is the difficulty in controlling the injection of environmentally induced faults. In addition, little is known about the relationship between faults in-jected in a laboratory and those actually occurring in the field.

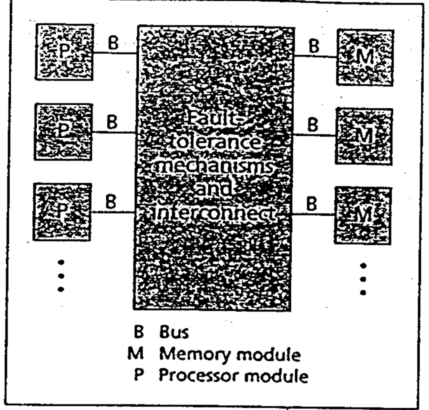


Figure 9. Class of architectures that React can analyze.

Finally, most fault-injection experiments are essentially case studies of particular systems. We must develop ways of generalizing machine-specific results to expand their applicability to other systems. Growing dependence on computers in life- and cost-critical applications makes this essential.

Acknowledgments

This material is partially based on work supported by the Texas Advanced Technology Program under grant 999903-029 and by the Air Force Office of Scientific Research under grant F49620-94-1-0276.

References

1. G.S. Choi and R.K. Iyer, "Focus: An Experimental Environment for Fault Sensitivity Analysis," *IEEE Trans. Computers*, Vol. 41, No. 12, Dec. 1992, pp. 1,515-1,526.
2. J. Karlsson et al., "Two Fault-Injection Techniques for Test of Fault-Handling Mechanisms," *Proc. Int'l Test Conf.*, IEEE CS Press, Los Alamitos, Calif., Order No. 2156, 1991, pp. 140-149.
3. R. Chillarege and R.K. Iyer, "An Experimental Study of Memory Fault Latency," *IEEE Trans. Computers*, Vol. 38, No. 6, June 1989, pp. 869-874.
4. R. Chillarege and R.K. Iyer, "Measurement-Based Analysis of Error Latency," *IEEE Trans. Computers*, Vol. C-36, No. 5, May 1987, pp. 529-537.
5. E.W. Czeck and D.P. Siewiorek, "Effects of Transient Gate-Level Faults on Program Behavior," *Proc. 20th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., Order No. 2051, 1990, pp. 236-243.
6. R. Chillarege and N.S. Bowen, "Understanding Large-System Failures: A Fault-Injection Experiment," *Proc. 19th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., Order No. 1959, 1989, pp. 356-363.
7. K.K. Goswami and R.K. Iyer, "A Simulation-Based Study of a Triple-Modular Redundant System Using Depend," *Proc. Fifth Int'l Conf. Fault-Tolerant Computing Systems*, IEEE Press, Piscataway, N.J., 1991, pp. 300-311.
8 J.A. Clark and D.K. Pradhan, "Reliability Analysis of Unidi-

rectional Voting TMR Systems Through Simulated Fault Injection," *Proc. 1992 Workshop Fault-Tolerant Parallel and Distributed Systems*, IEEE CS Press, Los Alamitos, Calif., Order No. 2871, 1992, pp. 72-81.

9. J. Arlat et al., "Fault Injection for Dependability Validation: A Methodology and Some Applications" *IEEE Trans. Software Engineering*, Vol. 16, No. 2, Feb. 1990, pp. 166-182.
10. Z. Segall et al., "Fiat: Fault-Injection-Based Automated Testing Environment," *Proc. 18th Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., Order No. 867, 1988, pp. 102-107.
11. G.A. Kanawati, N.A. Kanawati, and J.A. Abraham, "Ferrari: A Tool for the Validation of System Dependability Properties," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing*, IEEE CS Press, Los Alamitos, Calif., Order No. 2876, 1992, pp. 336-344.
12. J.A. Clark and D.K. Pradhan, "React: A Synthesis and Evaluation Tool for Fault-Tolerant Multiprocessor Architectures," *Proc. 1993 Annual Reliability and Maintainability Symp.*, IEEE Press, Piscataway, N.J., 1993, pp. 428-435.

***Jeffrey A. Clark** is a member of the technical staff in the Reliability and Maintainability Center at the Mitre Corporation. His research interests include fault-tolerant computing, parallel processing, and system dependability modeling and simulation. He received a BS in electrical and computer engineering (ECE) from Carnegie Mellon University in 1987, and an MS and PhD in ECE from the University of Massachusetts at Amherst in 1989 and 1993, respectively. He is a member of the IEEE Computer Society and the Reliability Society.*

***Dhiraj K. Pradhan** holds the College of Engineering Endowed Chair in computer science at Texas A&M University. He has been actively involved in research of fault-tolerant computing, parallel processing, and VLSI testing over the last 20 years, presenting and publishing numerous papers. He has served as guest editor for special issues on fault-tolerant computing in* IEEE Transactions on Computers *and* Computer, *and is an editor for several journals, including* IEEE Transactions on Computers *and* JETTA. *Pradhan has also served as general chair for the 22nd Fault-Tolerant Computing Symposium and as program chair for the IEEE VLSI Test Symposium. He is a fellow of the IEEE and a recipient of the Humboldt Distinguished Scientist Award.*

Readers can contact Clark at the Mitre Corporation, Mailstop H113, 202 Burlington Road, Bedford, MA 01730; e-mail jeclark@mbunix.mitre.org; and Pradhan at the Computer Science Department, H.R. Bright Building, Texas A&M University, College Station, TX 77843; e-mail pradhan@cs.tamu.edu.

III. Research in Mobile Computing

Providing Seamless Communications in Mobile Wireless Networks*

P. Krishna Bikram S. Bakshi N.H. Vaidya D.K. Pradhan
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112
Email : {pkrishna,bbakshi,vaidya,pradhan}@cs.tamu.edu

Abstract

This paper presents a technique to provide seamless communications in mobile wireless networks. The goal of seamless communication is to provide disruption free service to a mobile user. A disruption in service could occur due to active handoffs (handoffs during an active connection). Existing protocols either provide total guarantee for disruption free service incurring heavy network bandwidth usage (multicast based approach), or do not provide any guarantee for disruption free service (forwarding approach). There are many user applications that do not require a "total" guarantee for disruption free service but would also not tolerate very frequent disruptions. This paper proposes a novel staggered multicast approach which provides a probabilistic guarantee for disruption free service. The main advantage of the staggered multicast approach is that it exploits the performance guarantees provided by the multicast approach and also provides the much required savings in the static network bandwidth.

The problem of guaranteeing disruption free service to mobile users becomes more acute when the static backbone network does not use any packet numbering or does not provide retransmissions. Asynchronous Transfer Mode networks, the future of B-ISDN, display these properties. To make our study complete, we present a possible implementation of our scheme for wireless ATM networks.

1 Introduction

Mobility has opened up new vistas of research in networking. With the availability of wireless interface cards, mobile users are no longer required to remain confined within a static network premises to get network access. Users of portable computers would like to carry their laptops with them whenever they move from one place to another and yet maintain transparent network access through the wireless link. Integrated voice, data and image applications are going to be used by millions of people often moving in very heavy urban traffic conditions.

On the downside, mobility brings along with it a myriad of network management problems. The problems could be broadly classified as mobility management related and connection management related. In this paper we will primarily deal with a key problem in mobile wireless networks related to connection management. The problem deals with providing disruption free service to mobile users.

Future personal communication networks (PCN) will allow users to engage in bi-directional exchange of information including but not limited to voice, data, and image, irrespective of location and time, while permitting users to be mobile. Even though, near term personal communication services (PCS) are going to be voice-oriented, PCN are expected to support multimedia PCS in the long term [13]. This will spur requirements for high capacity wireless networks.

A typical PCN with mobile users [8, 9, 10] comprises of a static network and communication links between them. Some of the fixed hosts, called *base stations* (*BS*)[1] are augmented with a wireless interface and they provide a gateway for communication between the wireless and static network. Due to the limited range of wireless transreceivers, a mobile user can communicate with a *BS* only within a limited geographical region around it. This region is referred to as a base station's *cell*. A mobile user communicates with one *BS* at any given time. Each *BS* is responsible for forwarding data between the mobile user and the static network.

When a mobile host is engaged in a call or data transfer, it will frequently move out of the coverage area of the mobile support station it is communicating with, and unless the call is passed on to another cell, it will be lost. Thus, the task of forwarding data between the static network and the mobile user must be transferred to the new cell's ***mobile support station.*** This process, known as ***handoff***, is transparent to the mobile user. Handoff helps to maintain an end-to-end connectivity in the dynamically reconfigured network topology.

As the demand for services increase, the number of cells may become insufficient to provide the required quality of service. *Cell splitting* can then be used to increase the traffic handled in an area without increasing the bandwidth of the system. In future, the cells are expected to be very small (less than 50 meters in diameter) covering the interior of a building. The re-

*Research reported is supported in part by AFOSR under grant F49620-94-1-0276, and Texas Advanced Technology Program under grant 999903-029.

[1]Base stations are sometimes called *mobile support stations.*

duction in the cell size causes an increase in the number of handoffs, thereby increasing the signalling traffic (network load) due to the handoff protocol messages. In addition, handoff also causes a disruption in service if it is not done in a fast and efficient manner. In this paper we will primarily deal with design and implementation issues of handoff protocols to ensure disruption free service.

Providing connection-oriented services[14, 15, 16, 17, 18] to the mobile users requires that the user always be connected to the rest of the network in such a manner that its movements are transparent to the users. Providing disruption free service is a stronger requirement than mere connection-oriented services. In addition to maintaining the connection, the network will need to ensure that the delay experienced by the data packets over the network is less than a fixed time called the deadline. The deadline is in turn determined by the *quality of service* (QOS) required by the users. The goal of seamless communication is to provide disruption free service to a mobile user. A disruption in service could occur due to active handoffs (handoffs during an active connection). This is because traditional protocols require the old BS to forward data packets to the new BS. Thus, every time a mobile user moves into a new cell during the connection (active handoff), the user will see a break in service while the data gets forwarded to it from the old BS via the new BS.

We first present the proposed approach for providing disruption free service to mobile users. Our work differs from existing protocols in that the network load incurred by the proposed approach is significantly lower as compared to others. The number of disruptions seen by the user will depend on the number of handoffs incurred during the lifetime of the connection. The number of handoffs in turn depends on the mobility pattern of the user. In this paper we use two mobility models to analyze the proposed approach. In the first model, the user spends very little time in a cell (handoffs occur frequently), while in the other model the user spends a long time in a cell (handoffs occur infrequently). Analysis shows that for both these models, the proposed approach significantly reduces the network bandwidth usage without violating the quality of service (QOS) requirements specified by the user application.

The problem of guaranteeing disruption free service to mobile users becomes more acute when the static backbone network does not use any packet numbering or does not provide retransmissions. Asynchronous Transfer Mode networks, the future of B-ISDN, display these properties. The second half of the paper deals with implementation issues of the proposed approach. The backbone network has been assumed to be an asynchronous transfer mode (ATM) network. The vast transmission capacity offered by an ATM broadband network can provide communication services to a wide range of applications including video and audio. It is thus a natural choice for multimedia services. ATM is basically a connection-oriented switching technology. Users need to establish a fixed route called a *virtual channel (VC)* before any information can be exchanged. To make maximum use of available bandwidth, multiple *VC*s can be statistically multiplexed over the same link. Issues related to ATM have been comprehensively treated in [19, 22, 23].

While ATM promises to do away with the present problems faced by the telephony community, it raises a number of issues for the mobile computing industry. As mentioned before, existing ATM protocols do not offer any packet numbering and prohibit packet reordering. In this scenario maintaining a continuous (disruption free) communication link to the mobile host becomes complicated. We thus need to ensure that once a handoff takes place no packet is lost and deadlines are met, i.e., the packets that have been transmitted to the previous BS and which have not reached the mobile host due to handoff, are somehow delivered to it within the given time constraint. Keeping these problems in mind, we propose an easily implementable technique to provide disruption free service to mobile hosts in wireless ATM networks.

The rest of this paper is organised as follows. In section 2 we briefly review related work. The basic idea behind our scheme is presented in section 3. Section 4 presents the issues related to implementation of the proposed approach using ATM as the backbone network. Concluding remarks are presented in section 5.

2 Related Literature

Keeton et al in [2] proposed a set of algorithms to provide connection oriented network services to mobile hosts for real time applications like multimedia. Their solutions lay excellent groundwork for work in this area but did not guarantee disruption free service. In fact their scheme was shown to suffer from extended intervals of time when service to the mobile host was disrupted. A study done in [1] shows that if the handoff protocol required forwarding data between the BSs connected by physical links, then a high bandwidth (between 48Mbps and 96Mbps) is required just to forward these data packets. Moreover, loops can be formed in the connection path if forwarding is employed. This will lead to inefficient network utilization.

A multicast based solution was proposed in [1]. In this approach, the data packets for a mobile host are multicast to the BSs of the neighboring cells so that when the host moves to a new cell, there are data packets already waiting for it and thus, there is no break in service. It is evident, however, that this scheme is not cost effective. As the number of users in the network increases, the amount of network bandwidth used up by the multicast connections is going be prohibitively high. In [1], the cost of such a multicast scheme was determined to be the buffer overhead at the BSs. Our view of the problem is that the major component of cost incurred in a multicast based approach will be the *amount of extra bandwidth used*, and not the buffer overhead at each BS. This argument is supported by the availability of cheap memory but expensive network bandwidth[2].

[2]The cost of a 30 minute call from USA to Japan is approx-

As pointed out in [3], the *network call processor*[3] in a static network becomes the bottleneck in an environment where handoffs are frequent and require excessive interaction with a base station – an inherent problem associated with the work in [2]. To alleviate this problem, the authors in [3] proposed a new network architecture which made use of *virtual circuit trees* to minimize handoff processing. However, it does not discuss about providing disruption free service when a handoff takes place – the mainstay of applications like multimedia [23].

We find that while existing literature is a rich source of protocols and models for tackling the problem in hand, there does not exist a cost-effective solution for providing disruption free service.

3 Proposed Approach

Traditional multicast-based schemes require the packets to be multicast throughout the length of the connection. This leads to wastage of network bandwidth. The communication links from the switch to the *BS*s other than the *BS* of the cell where the mobile host is currently located get unnecessarily loaded. As the number of mobile hosts increase in a cell, the total network usage due to multicast connection for each host will become enormous. Due to this extra network usage, new connections might be blocked because the network capacity is exceeded.

The thrust of our approach is to avoid unnecessary multicast. A multicast throughout the length of the connection may prove to be unecessary if the network had some information – e.g., how long is the mobile host going to remain in the same cell (this period is called *cell latency*). If the network has such information, then the multicast need not be done during that period of time.

The main idea of the proposed approach is to "stagger" the multicast initiation by the amount of time one is sure that the host remains within a cell, i.e., for a time interval equal to the *cell latency*. The *cell latency* will solely depend on the mobility model of the host. In this paper we will analyze the proposed approach based on two mobility models. One model is *pessimistic* in nature, and the other *optimistic*. By *pessimistic* we mean that the *cell latency* for a mobile host is very small. On the other hand in the *optimistic* model, the mobile host remains in a cell for a longer time.

We will now present the staggered multicast approach.

3.1 Staggered Multicast

If the mobility pattern of a user could be modelled in such a way that it can be ascertained with a certain probability that the user is going to remain in the same cell for t_s amount of time, multicast could be avoided for this amount of time. The value of t_s then[4] gives us a measure of the stagger time than can be safely introduced before initiating a multicast. This way, we will save on the network usage, and still guarantee disruption-free service with a certain probability.

Let P_i be the probability of disruption during the *i-th* handoff, and t_i be the *cell latency* before the *i-th* handoff. Let t_{mi} be the time spent in multicast mode before the *i-th* handoff. A disruption occurs when a mobile host initiates a handoff before multicast has been initiated. Then the probability of disruption during the *i-th* handoff can be given as,

$$P_i = Pr[t_s > t_i]$$

Let the number of handoffs occuring over the length of the connection time T_c be N_h. Let $P_{disrupt}$ be the average probability of disruption during a handoff. $P_{disrupt}$ is determined as,

$$P_{disrupt} = \frac{1}{N_h}\sum_{i=1}^{N_h} P_i$$

The value of $P_{disrupt}$ can now be used as a measure of the Quality of Service (QOS). There are a number of applications that cannot tolerate disruptions during the time of connection, i.e., $P_{disrupt} = 0$. Two existing examples of such applications are *telemedicine*, and *video conferencing*. With increased availability of mobile computing applications, a large number of hitherto unexplored applications will emerge. The applications mentioned here are but only a small sample.

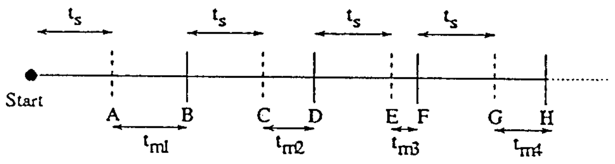


Figure 1: Total Guarantee

Figure 1 presents an example showing the times of handoffs and multicast initiations. The times B, D, F, and H represent the time at which handoff takes place. The times A, C, E, and G represent the time at which multicast is initiated. The cell latencies for Figure 1 are $t_1 = t_s + t_{m1}$, $t_2 = t_s + t_{m2}$, and so on. For total guarantee, the following should hold.

$$\forall i, 1 \le i \le N_h, t_s < t_i$$

i.e., for all handoffs a multicast is initiated within the associated *cell latency* interval.

However, there are a lot of applications that do not have a strict requirement of disruption free service during every handoff. A probabilistic guarantee

imately equal to the cost of 1 Mbyte of RAM.

[3]The role of the *network call processor* is to establish a path or route at connection setup time. While doing so it takes into account the network load so as to balance the load on each network node.

[4]It should be mentioned here that the actual time for stagger is less than t_s, because the time to set up the multicast connections should be taken into account. This has been dealt in greater detail in Section 4.3 for a wireless ATM network.

is sufficient for such applications, i.e., $P_{disrupt} \geq 0$. Examples of such applications include *ftp*, *audio channels* and *movies*. If the QOS requirement can be expressed as a probablisitic guarantee for disruption free service, then the multicast initiation could be further staggered resulting in an even greater reduction of network usage.

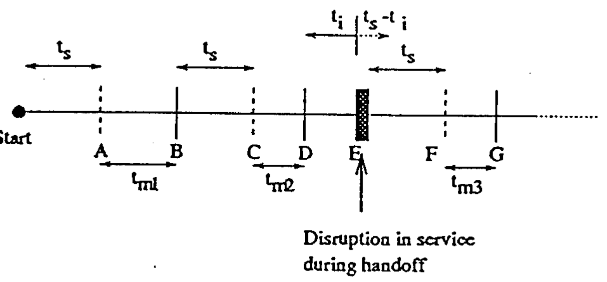


Figure 2: Probabilistic Guarantee

To illustrate this probabilistic scheme we present an example as shown in Figure 2. This figure shows the times of handoffs and multicast initiations in the multicast scheme that provides a probabilistic guarantee. The times B, D, E, and G represent the time at which handoff takes place. The times A, C, and F represent the time at which the multicast is initiated. As noticed in the figure, there is a disruption in service during handoff at time E, because, there was no multicast initiated before the handoff. Thus, a disruption occurs during the *i-th* handoff when the stagger time t_s is greater than the cell latency time t_i.

In the absence of any empirical data for user mobility, we propose to evaluate the effectiveness of our scheme using two mobility models, which we believe cover a wide range of user mobility. At this point we would like to mention that the main aim of this paper is not to show that the two models cover the whole spectrum of user mobility, but, to show that with the aid of user mobility information, we can drastically reduce the network load and still provide disruption-free service. We will be able to correctly estimate the benefits obtained from the proposed approach only if we can accurately model the user mobility.

3.2 Mobility Models

3.2.1 *Optimistic* Model

The *optimistic* model is based on the two dimensional random walk model. Let the two dimensions be the X axis and the Y axis. In such a model, the user tosses two coins every T seconds. Based on the resulting head-tail combination, the user will decide to take a step of size s meters in a specific direction (e.g., head-head results in a step in the north-east direction). Let the distance of the user with respect to the center of the circular cell at time t be $r(t)$. As derived in Appendix 1, the probability that a mobile user will remain in the same cell at time t is given as,

$$Prob(r(t) < R) = 1 - e^{-\frac{R^2}{2\alpha t}} \qquad (1)$$

where, R is radius of the cell, and $\alpha = s^2/T$.

Let us consider a picocellular environment, which is more suited for pedestrian traffic. Let $s = 0.4$m, $T = 0.25$ sec. Therefore, $\alpha = 1.0$. We vary the radius of the circular cell R from 10m to 50m. The variation of the probability with time is illustrated in Figure 3. As seen in the figure, the probability of the mobile

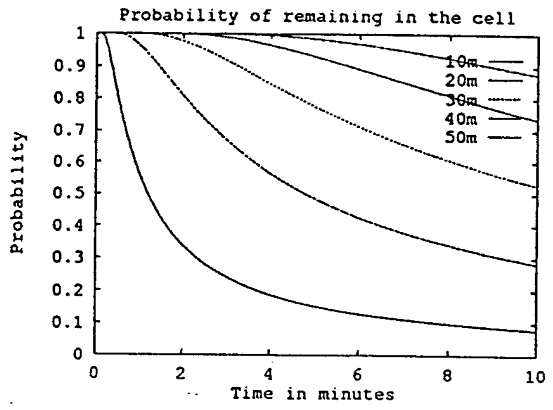


Figure 3: Probability of being in a cell

host remaining in a cell decreases with time. An interesting observation however is that even after 3.25 minutes (195 seconds), the probability that the user is still in the same cell ($R = 30m$) is as high as 90%. This mobility model represents the class of users who spend a lot of time in a cell.

3.2.2 *Pessimistic* Model

The *pessimistic* model is based on the mobility model proposed in [5]. In this model the mobile user is assumed to be moving at an average velocity of V. The direction of movement is uniformly distributed over $[0, 2\pi]$. The mobile users are assumed to be uniformly distributed over the cell area with a density of ρ. If the length of the cell boundary is L, and the cell area S, the number of mobile users crossing the cell boundary per unit time is given by $\frac{V\rho L}{\pi}$. If ρ can be assumed to remain constant over the entire cell area, the average cell crossing rate of a mobile user is given by $\frac{VL}{\pi S}$. For circular cells, L will correspond to the perimeter of a cell, and thus $\frac{L}{S} = \frac{2}{R}$. It follows that the average *cell latency* of a mobile user is given $\frac{\pi R}{2V}$. As in [5], we will assume that the *cell latency* of a mobile user is exponentially distributed with a mean $\frac{\pi R}{2V}$.

3.3 Performance Analysis of the Staggered Multicast Approach

The overhead of the staggered multicast scheme can be characterized by the total time spent in the multicast mode T_m as compared to the length of connection T_c. T_m is determined as

$$\sum_{i=1}^{N_h} t_{mi}$$

where, t_{mi} is the time spent in multicast mode before the i-th handoff, and N_h is the number of handoffs occuring over the length of connection. The total time spent in the unicast mode, T_u, is then given by the difference, $T_c - T_m$. We determine the overhead of the multicast scheme as the fraction of the total connection time spent in the multicast mode,

$$Overhead = \frac{T_m}{T_c}$$

The QOS measure of the staggered scheme (characterized by $P_{disrupt}$) is now given as

$$QOS = 1 - P_{disrupt}$$

3.3.1 Performance of *Optimistic* Model

In this section we present the results of the staggered multicast scheme obtained using the *optimistic* model. We performed simulations to analyze the staggered multicast scheme. The radius of the circular cell R was varied from 10m to 50m. The time of connection T_c, was fixed to be 100 minutes. The step size s was chosen to be 0.4m, and the time interval between two tosses T was chosen to be 0.25 s.

As stated earlier, we characterize the overhead as T_m/T_c. Figure 4 illustrates the variation of overhead with the stagger time t_s. It is noticed in Figure 4 that the overhead reduces as the stagger time increases. This is because as stagger time increases, the amount of time spent in multicast mode reduces. Thus, the overhead, determined as T_m/T_c, reduces. It can also be noticed that for a given stagger time, the overhead increases with an increase in cell radius. This is because as the radius increases, the time interval between handoffs increases. If stagger time is kept constant, we are not making use of the potentially extra time available due to increased cell radius. As a result the fraction of time spent in multicast mode increases.

We also evaluated the probability of a disruption during a handoff, $P_{disrupt}$. As stated earlier, a disruption occurs only if multicast is not initiated before a handoff occurs. Figure 5 illustrates the variation of probability of disruption with stagger time t_s. Higher the stagger time, higher is the probability of disruptions. It can also be noticed that for a given stagger time, the probability of disruption increases as the radius of the cell decreases. This is because as the radius decreases, the probability of remaining in a cell reduces for a given stagger time. Therefore, the probability of disruption increases.

Using these results, the network can determine the appropriate stagger time for a user. Let us illustrate it with an example. Let the radius of the cells in the network be 30 m.

Suppose that the users in a network are maintaining non-critical connections. This means that a probabilistic guarantee will suffice. Let the QOS demanded by the users be 75 %. This means that $P_{disrupt} = 25\%$, i.e., on an average three out of four handoffs will be guaranteed to be disruption free. Then, using Figure 5, we can determine the appropriate stagger time,

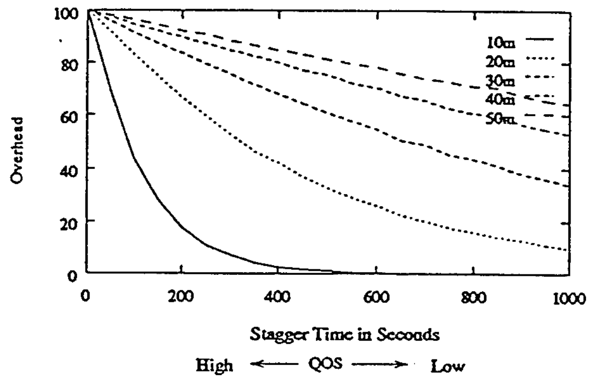


Figure 4: *Optimistic* Model : Overhead Vs Stagger Time

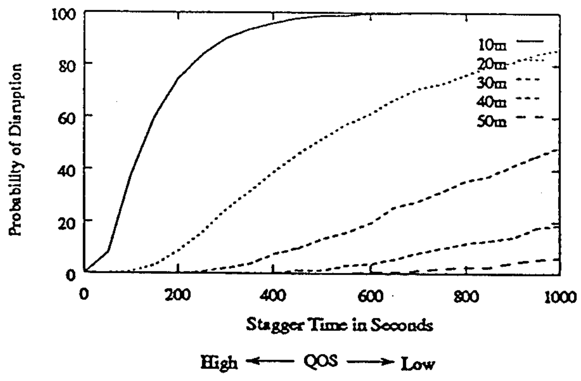


Figure 5: *Optimistic* Model : Probability of Disruption Vs Stagger Time

which is 650 seconds (approx. 11 minutes). Therefore, the multicast initiation can stagger by 11 minutes and we will still provide the desired QOS to the users. The overhead of such a staggered scheme can be determined using Figure 4 to be 50%. Therefore, the network spends only 50% of the total connection time in multicast mode for the user. In a traditional multicast based solution for disruption free service, the network spends 100% of the connection time in multicast mode [1]. Comparing it to the traditional multicast based solutions, there is a 50% savings in network bandwidth.

Suppose on the other hand, a user is maintaining a critical connection which demands total guarantee of disruption free service, i.e., the QOS demanded by the user is 100 %. Even though a non-zero value of stagger time could be obtained for $P_{disrupt} = 0$ in the *optimistic* model (e.g., $t_s = 3$ minutes for $R = 30m$ in Figure 5), this may not be true in general for other models. In fact the next model shows that for total guarantee of disruption free service, stagger time has to be zero. In other words, for total guarantee of disruption free service, multicast should be done throughout the length of the connection.

3.3.2 Performance of *Pessimistic* Model

Simulations were performed to analyze the multicast scheme using the *pessimistic* mobility model. The mobility model for this part was same as the mobility model proposed in [5]. The radius of the circular cell R was varied from 10m to 50m. The time of connection T_c, was fixed to be 100 minutes. The average velocity V was chosen to be 1.6 m/s (approx 5.7 km/hr, for a pedestrian user).

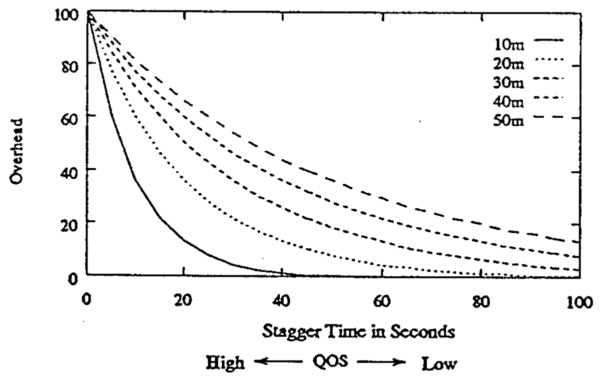


Figure 6: *Pessimistic* Model : Overhead Vs Stagger Time

The trends in the variation of overhead (Figure 6) and probability disruption (Figure 7) with respect to stagger time for the pessimistic mobility model are similar to the optimisitic model. But as was expected, the allowable stagger time in the pessimistic model for a particular QOS is very low compared to the allowable stagger time in the optimistic model. For example, when $R = 30$m, QOS = 75 %, the allowable stagger time in the optimistic model is 650 seconds. On the other hand for a pessimistic model, the allowable stagger time is only 9 seconds.

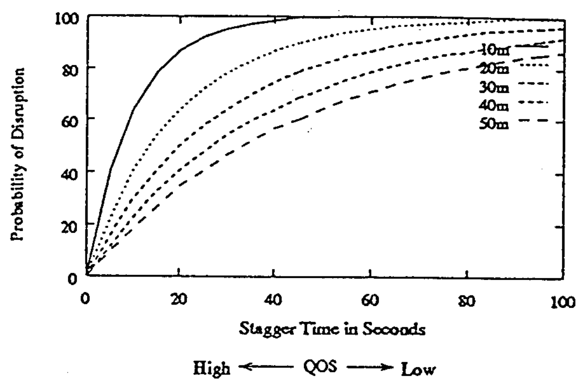


Figure 7: *Pessimistic* Model : Probability of Disruption Vs Stagger Time

Another noticeable difference with the optimistic model is that there is no stagger time allowable for total guarantee service (i.e., when $P_{disrupt} = 0$). Thus, for a user whose mobility pattern can be modelled with the pessimistic model, multicast has to be done throughout the connection time if the user desires total guarantee of disruption free service. On the other hand, if the user requires only a probabilistic guarantee, then a non-zero stagger can be introduced. For example, if the QOS demanded by the user is 75 %. Then, using Figure 7, we can determine the allowable stagger time to be 9 seconds. The overhead of such a staggered scheme can be determined using Figure 6 to be 73 %. Therefore, when compared to the traditional multicast schemes, there is a 27% savings in network bandwidth.

3.3.3 Discussion

In this section we have presented a staggered multicast approach. The main features of this approach are that it saves network bandwidth by providing a probabilistic guarantee for disruption free service. We analyzed the proposed approach for two mobility models. These models represented two different classes of mobile users – those with high *cell latency*, and those with low *cell latency*. The results indicate that regardless of the mobility model, the proposed approach provides tremendous savings in network bandwidth for applications that require a probabilistic guarantee. We expect the performance of the proposed approach for a typical user mobility model to lie somewhere in between the performance gains obtained for these models.

In the next section we will present an implementation of the proposed approach on a wireless ATM network.

4 Implementation on ATM Network

4.1 System Model

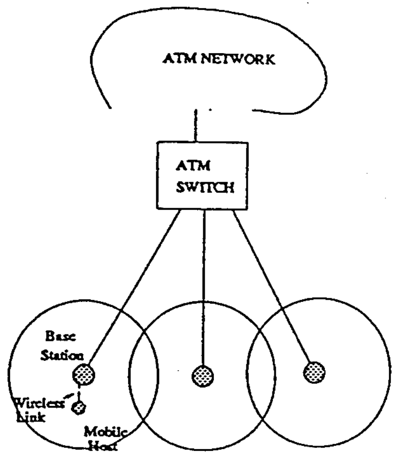


Figure 8: PCN Model

We view the future personal communication network as a two tier network - a backbone static ATM network and a peripheral wireless network. This model is similar to the one proposed in [4]. Figure 8 shows ATM switches connected to base stations which in turn provide service to the mobile hosts. ATM cells are received by the base stations from the static network and forwarded to the mobile hosts.

4.2 Protocol

We define a multicast group g_i as the set of base stations that are included in the multicast operation for the mobile host i. The base stations maintain a table which maps each mobile host in its cell to its multicast group members. The group members for a mobile host can be determined based on some hints (direction, velocity). If no hints are available, the default multicast group members will be the neighboring base stations [1].

The connection management problem can be divided into two phases, namely, connection establishment phase and connection maintenance phase. The source mobile host initiates the connection establishment phase by sending a *connection request* message to its base station. The base station forwards this message to its switch. The switch assigns a VCN (virtual circuit number) for the source mobile host. The switch then initiates a *locate* procedure for the destination mobile host [10, 11, 12]. Upon getting the location information of the destination mobile host, a connection is set up between the source and the destination mobile host via the switches at the source and the destination.

Our work differs in the connection maintenance phase. Please refer Figure 9 for the discussion. The thick lines in Figure 9 represent the data packets being transferred over the static network, and the thick dashed lines represent the data packets being transferred over the wireless medium between the base station and the mobile host. The thin lines represent the control messages being transferred over the static network, and the thin dashed lines represent the control

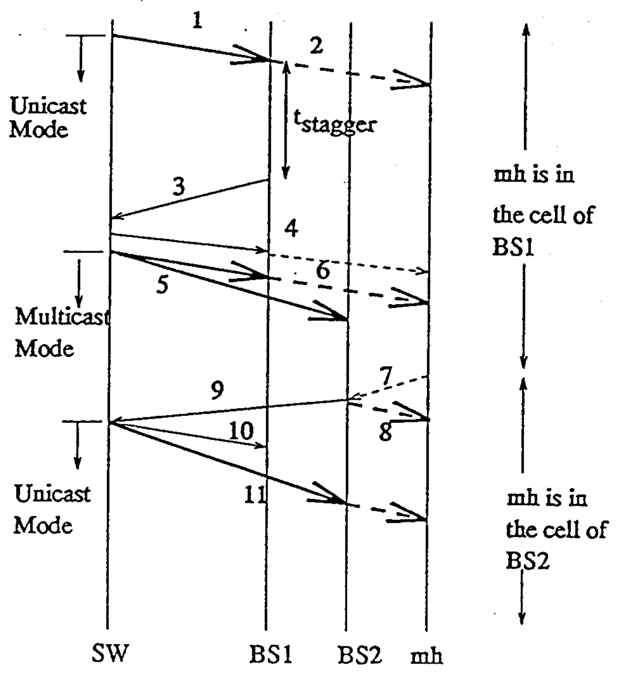


Figure 9: Connection Maintenance

messages being transferred over the wireless medium.

Once a connection is established, the switch SW is in the unicast mode, i.e., it forwards the data packets to only the "current" base station $BS1$, which in turn forwards it to the mobile host mh (steps 1-2). After $t_{stagger}$ units of time[5], $BS1$ sends a *multicast initiate* message to SW (step 3). The multicast group members g_{mh} are tagged along with the message. The switch SW then determines the crossover point for the multicast group members. The VCNs to the base stations in g_{mh} are assigned, and the switch SW sends back the list of VCNs to $BS1$ which forwards it to mh (step 4). Upon receiving an acknowledgment from mh, SW enters the multicast mode. Let us suppose that the multicast group members are $BS1$ and $BS2$. SW multicasts the data packets to the base stations $BS1$ and $BS2$ (step 5). However, only the current base station which is $BS1$ forwards the data packets over the wireless medium to mh (step 6). This continues till the mobile host mh detects that it has to handoff to $BS2$. The mobile host mh then sends a *handoff initiate* to the new base station $BS2$ (step 7). The base station $BS2$ starts transmitting data to the mobile host (step 8). It also forwards *handoff initiate* message to the switch SW (step 9). The switch SW then terminates the connections to the multicast group members except for $BS2$ (step 10). SW then reenters the unicast mode and sends the data packets to only the "current" base station $BS2$, which in turn forwards it to mh (step 11).

[5]It will be shown later that $t_{stagger} < t_s$, where t_s is the stagger time derived in Section 3.

4.3 Implementation Issues

Given the lossy nature of the wireless medium, there may be a need to frame groups of ATM cells at the BS and assign them some kind of sequence numbers. Additional bits to enable error correction and to allow recovery schemes may also be required for each frame. Likewise communication from the mobile host to a base station will consist of frames of ATM cells with additional bits as described above. Going by the philosophy behind ATM, it is likely that each frame will be small and of equal size. In line with this, we assume that all frames will be of fixed length containing F ATM cells.

Before we can apply our scheme to an ATM environment, we must take into account the various properties of ATM network protocols that make them differ from existing network protocols. As was mentioned before, ATM is a connection oriented switching technology where connections must be established for the entire duration of the call. Connection establishment consists of assigning a VCN (virtual circuit number) and/or a VPN (virtual path number), and allocation of resources both within the network and at the source and destination to support this connection. In a mobile environment we will thus need to ensure that before a mobile host hands off, connection has already been established between the new base station and the destination. For this purpose, we make use of a *dynamic virtual connection tree (dvct)* based network architecture, an extension to the idea proposed in [3]. For sake of completeness, we will describe the virtual connection tree in some more detail.

A virtual connection tree [3] is a set of cellular ATM switches and base stations in the static network that are chosen at call setup time to route ATM cells. The network is divided into *neighboring access regions* and the mobile host is assigned a set of *VCNs*, one for each base station in this region. As soon as the mobile host detects that it is entering another wireless cell, it starts transmitting its messages with the VCN assigned for that base station. This change in position of the mobile host is updated at the root of the virtual connection tree (an ATM switch that maintains the routing tables for this connection) as soon as the first ATM cell from the mobile host arrives bearing the new VCN. The study showed considerable reduction in load on the network call processor. The only time that the network call processor participates in a handoff is when the mobile host changes its neighboring access region. As noted by the authors, handovers within this connection tree are handled entirely by the mobile itself in a totally distributed fashion.

A *dvct* differs from a virtual connection tree in that the choice of participating base stations and ATM switches depends on the current location of the endpoints and may change *dynamically*, i.e., base stations and switches may be dynamically added and removed depending on the movement of the mobile host. All the base stations and ATM switches included in the multicast operation can now be viewed as a *dvct*. Figure 13 is an example of how bidirectional communication takes place between two end points – both of which may be mobile, in a *dvct* using the multicasting approach.

We consider an example to make the *dvct* approach more clear. Suppose that switch A is connected to base stations a and b. Let a be providing a connection between a mobile host $m1$ in its wireless cell and a mobile host $m2$ in the wireless cell of BS d which is connected to switch D. The table shown in Figure 13 represents the routing information maintained by switch A. Such information is present at all switches in the ATM network. Data coming out of host $m1$ carries VC1 (for BS a) which was assigned at connection set up time. Switch A translates VC1 to VC2 after a look up of its routing table and sends out this data through port 2. Switch B further translates the header information so that it now carries the VC3. Finally switch D translates this to VC5 before passing it on to the BS d and from there to host $m2$. On the return path, $m2$ sends out data carrying the VC7. Suppose the multicast group members for mobile host $m1$ are base stations a, b and c. Then, switch D translates VC7 to VC8 followed by translation to VC13 (and VC9 for multicast) at switch B. Finally switch A translates VC13 to VC14 (and VC15 for multicast) for the multicast members a and b. The onward transmission to host $m1$ is done by a. Now if host $m1$ hands off to base station b, it will continue normal transmission but with VC16, and continue receiving with VC15. It is easy to see that allocating VCNs to all base stations that are included in a multicast, will greatly ease the handoff process.

The total delay experienced by an ATM cell over the network can be characterized by two main components [23, 22].

$$\tau_{delay} = D_{cons} + D_{var} \qquad (2)$$

where D_{cons} represents the constant component and D_{var} represents the variable component of the delay. D_{cons} depends on the the physical delay of the medium and the distance an ATM cell has to travel between source and destination. D_{var} on the other hand is representative of the variation in queueing delays experienced by different ATM cells over the same connection in the network. Given the nature of delay variation experienced by different ATM cells, it is easy to see that different cells may experience different total delays over the same connection. This variation in cell delay is also referred to as *jitter*. [22] presents delay and delay variation objectives for two-way session audio and video services.

Crossover points within the network have significance when multiple connections are branching off from a common stream. Each connection in a multicast operation need not start from the source but may in fact find an intermediate switch that is handling the connection for some other base station (See Figure 11 for an example of crossover point location during handoff.). We model the delay experienced by an ATM cell over different routes starting from the crossover point to be bounded by the times τ_{min} and τ_{max}. The delay variation for each connection may now be viewed simply by a *delay pipe* as shown in Figure 10. The tail of the pipe represents the entry point of an ATM cell from the source.

It is evident from Figure 10 that at any given instant for a multicast operation, the tail of each pipe contains the same ATM cell. However, due to different delays experienced on different routes, the ATM cells coming out from the heads of different pipes to the respective base stations may not be the same.

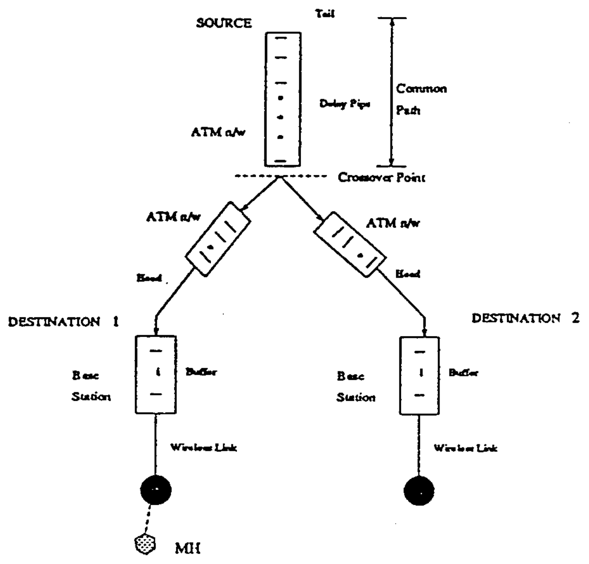


Figure 10: Delay Pipe Model

If the mobile host is to get consistent information from a base station during and after handoffs, then we have to make sure that the base stations involved in the handoff procedure have corresponding ATM cells in their buffers. Using this information and from the discussion above, it follows that the buffer requirement for ensuring that consistent information is present at the participating base stations is given by

$$Buffersize \geq BW_{conn} \times (\tau_{max} - \tau_{min}) \qquad (3)$$

where BW_{conn} is the bandwidth of the connection.

The ATM cell stream originating at the source consists of cells arriving back to back with no way of differentiating between two data cells. Of course, special cells may be generated by setting the appropriate bits in their headers, but this is not the case with data cells in particular. Extensions to existing ATM protocols to suit the mobile environment are discussed in [4]. However, our solution does not require any changes in existing protocols but targets ATM switch fabrics to achieve its goals.

In Figure 11 , BS1 is the base station that is currently transmitting to the mobile host and BS2 is the base station that is required to join the multicast. After waiting for time $t_{stagger}$, BS1 sends out a request to the switch to include the base stations in the multicast group of MH (g_{MH}) in the multicast operation. The upper bound on time taken for this is represented by t_{setup}. Note that t_{setup} includes the time required to

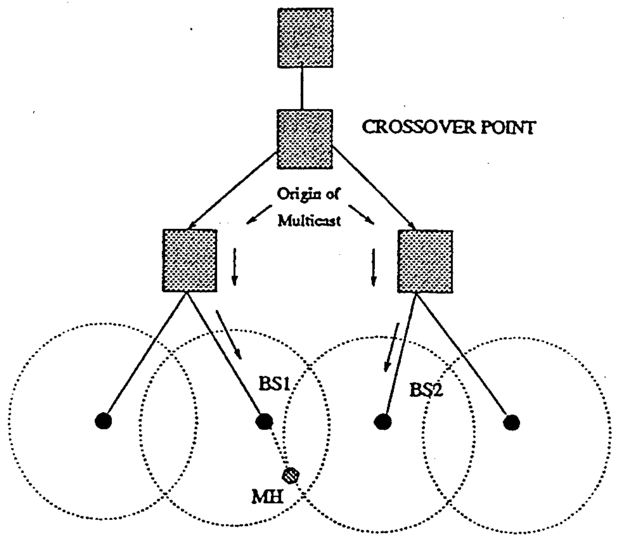


Figure 11: Handoff between Base Stations

- find the crossover point between BS1 and the base station farthest (in terms of number of intermediate switches) from it (BS2 in Figure 11),
- to update the multicast table entries in the crossover switch and
- to send the newly allocated VCNs of each base station to the mobile host.

As mentioned earlier, each time the mobile host performs a handoff[6] it is necessary to ensure that the sequence of frames being received from the old and new base stations preserve their relative order. We propose to overcome this problem by generating a control cell at the crossover switch when a new multicast connection is admitted. This control cell will act as a reference point within the ATM cell stream to faciliatate framing at each base station.

Implementation of our scheme will require minor modifications at the switch level. We would require the mobile host to maintain some kind of a record of the last frame number correctly received from a base station. A representative switch fabric that supports multicast (broadcast) [21] is shown in Figure 14. The modifications proposed to this switch architecture, however, are general enough to be applied to any other existing architecture. Our purpose is only to demonstrate how our scheme can be implemented. In the original switch architecture, CP is responsible for establishing both point-to-point and multicast connections. CN makes copies of the incoming ATM cells while the BGTs fill out the header information for each ATM cell generated by the CN (for multicast) as well as perform header translation for unicast cells. The DN distributes traffic over its outlets as uniformly

[6]Note that both BS1 and BS2 may not be connected to the same ATM switch. In fact the crossover point could require a number of hops to be made.

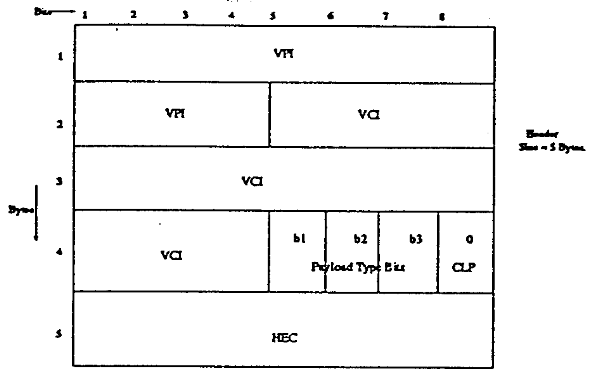


Figure 12: Possible header of a Control Cell

as possible. For a comprehensive survey on switch architectures see [23, 22].

The modifications required are shown with dashed lines in Figure 14. On receiving a multicast join request, the CCGL will request the CN to generate an empty cell of 48 bytes while the CP is setting up the multicast connection. A BGT will then attach the header of this control cell and appropriate values of *Cell Loss Priority (CLP)* and *Payload type (PT)* bits will be filled in. Note that the control cell will have its CLP bit set to 0 (high priority) and PT bits (b1, b2 and b3) may be set such that this control cell is distinguished from ordinary data cells. The VPI and VCI bits will be identical to their counterparts in the data cells. Figure 12 depicts a possible header configuration for a control cell.

When BS1 receives this control packet, say *cp* (which may be in the midst of regular cells all belonging to a single frame), it continues its framing process as before but sets a special flag in this frame before it goes out to the mobile host. The next frame to be transmitted will be numbered 1. The special flag that was set in the last frame to be transmitted will cause the mobile host to reset the frame counter it maintains to 0. Note that the mobile host does so only after it has received all previous frames from BS1 correctly. This will ensure that there is no confusion if requests for retransmission are generated by the mobile host on account of erroneous transmission from BS1. On receipt of *cp*, BS2 starts framing ATM cells (F cells per frame) and also starts numbering them from 1.

When handoff actually takes place, the mobile host will be able to specify the last frame completely received from BS1 (say n) so that BS2 can can send the next appropriate frame to the mobile host. If BS2 starts transmitting from frame $n+1$ onwards, it may result in one frame being completely duplicated at the mobile host in the worst case. The extent of duplication depends on the position of *cp* relative to the boundary of a frame being generated at BS1. Note that this duplication could be as small as a single ATM cell if the wireless protocol adopted transmits one ATM cell at a time instead of a lårger frame. In any event loss of ATM cells will not occur.

Given below is the expression for the synchonization time (T_{synch}) required to ensure that duplication in receiving ATM cells at the mobile host is limited to at most one frame.

$$\begin{aligned} T_{synch} &= t_{setup} + 2\tau_{max} - \tau_{min} + \tau_{bs2} \\ &\quad + \frac{F}{BW_{WL}} + \tau_{WLL} \end{aligned} \tag{4}$$

where BW_{WL} is the bandwidth of the wireless link, and τ_{WLL} is the latency of the wireless link. τ_{max} represents the upper bound on the time taken for the first frame to be reach BS2. An additional $(\tau_{max} - \tau_{min})$ represents the upper bound on the time required to flush out the ATM cells which was already received by BS1 before *cp* arrived. The expressions τ_{bs2} and $\frac{F}{BW_{WL}}$ represent the processing time required for a frame at BS2 and the time required to transmit a frame over the wireless link respectively.

Note that this analysis assumes that the delay associated with ATM cells reaching $BS1$ is τ_{min}, and the delay for ATM cells reaching $BS2$ is τ_{max}. This analysis will produce the worst case value of T_{synch}.

The actual stagger time available for this connection is now given by

$$t_{stagger} = t_s - T_{synch} \tag{5}$$

where, t_s is the stagger time determined for a particular QOS requirement of the user (see Section 3).

Note that the synchronization time presented above and the chosen buffer size of $(\tau_{max} - \tau_{min}) \times BW_{conn}$ at each base station, will together ensure that the frame being currently transmitted to the mobile host is within the buffer for each base station in the multicast.

The scheme presented here may result in the duplication of a single frame of ATM cells as explained above. However, this duplication could be as small as a single ATM cell if the wireless protocol adopted transmits one ATM cell at a time instead of a larger frame. It is possible to avoid any duplication if control cells can be generated at the source itself. However, this may require a change in the existing ATM protocols. In this paper, we do not consider such a situation but it is evident that if such a change is brought about in the future, then we will be able to perfectly synchronize frame reception at the mobile host.

5 Conclusion

There are many user applications that do not require a "total" guarantee for disruption free service but would also not tolerate very frequent disruptions. An user will not not want to pay a high cost for such applications. Thus if a multicast based approach is used, the data packets will be multicast to the neighboring wireless cells throughout the connection. This will be prohibitively expensive. On the other hand, if forwarding is used during handoffs, the user will see a break in service during every handoff. With the decreasing cell sizes, the user might see a disruption every 5 seconds (in picocellular environments). Proposed in this paper is a novel staggered multicast approach which provides *probabilistic* guarantee for disruption free service. The main advantage of the staggered multicast approach is that it partially provides the benefits of the multicast approach and also provides the much required savings in the static network bandwidth.

In summary, the main features of the staggered multicast approach are the following:

- The network bandwidth usage is significantly reduced.
- A probabilistic guarantee for disruption free service is provided.

Using the ATM switch modifications as suggested in the implementation section of the paper, we can ensure lossless data delivery to the end user.

We are currently investigating staggered multicast schemes where the stagger time is determined dynamically during the handoff process. We believe that a dynamic stagger will more provide a much better performance than the static stagger scheme proposed in this paper. On the other hand, if there are sophisticated wireless adapters available that can provide an intermediate signal level which will notify the mobile host that a handoff will soon occur, then the multicast initiation could be staggered till this point.

References

[1] R. Ghai and S. Singh, "An Architecture and Communication Protocol for Picocellular Networks," *IEEE Personal Communications Magazine*, pp. 36-46, Vol.1(3), 1994.

[2] K. Keeton et.al., "Providing connection-oriented network services to mobile hosts," *Proc. of the USENIX Symposium on Mobile and Location-Independent Computing*, Cambridge, Massachussets, August 1993.

[3] Anthony S Acampora, Mahmoud Nagshineh, "An Architecture and Methodology for Mobile-Executed Handoff in cellular ATM Networks," *IEEE Journal on Selected Areas on Communications*, October, 1994.

[4] D Raychauduri and N Wilson, "ATM Based Transport Architecture for Multiservices Wireless Personal Communication Networks." *IEEE Journal on Selected Areas on Communications*, October 1994.

[5] R. Thomas, H. Gilbert, and G. Mazziotto, "Influence of the Mobile Station on the Performance of a Radio Mobile Cellular Network," *Proc. 3rd Nordic Sem.*, paper 9.4, Copenhagen, Denmark, Sep., 1988.

[6] A. Papoulis, "Probability, Random Variables, and Stochastic Processes," Third Edition, McGraw-Hill, Inc.

[7] E. Kreyszig, "Advanced Engineering Mathematics," Fifth Edition, John Wiley & Sons, Inc.

[8] W. C. Y. Lee, *Mobile Cellular Communications Systems*, McGraw Hill, 1989.

[9] D. M. Balston and R. C. V. Macario, *Cellular Radio Systems*, Artech House, 1994.

[10] S. Mohan and R. Jain, "Two User Location Strategies for Personal Communication Services," *IEEE Personal Communications*, Vol. 1, No. 1, 1994.

[11] P. Krishna, N. H. Vaidya and D. K. Pradhan, "Location Management in Distributed Mobile Environments," *Proc. of the Third Intl. Conf. on Parallel and Distributed Information Systems*, pp. 81-89, Sep. 1994.

[12] P. Krishna, N. H. Vaidya and D. K. Pradhan, "Efficient Location Management in Mobile Wireless Networks," Technical Report, Dept. of Computer Science, Texas A&M University, Feb., 1995.

[13] C. Lo and R. Wolff, "Estimated Network Database Transaction Volume to Support Wireless Personal Data Communications Applications," *Proc. of Intl. Conf. Communications*, May, 1993.

[14] Pravin Bhagwat and Charles. E. Perkins, "A Mobile Networking System based on Internet Protocol (IP)," *Proc. of the USENIX Symposium on Mobile and Location-Independent Computing*, Cambridge, Massachussets, August 1993.

[15] J. Ioannidis et. al., "IP-based Protocols for Mobile Internetworking," *Proc. of ACM SIGCOMM*, 1991.

[16] J. Ioannidis and G. Q. Maguire Jr., "The Design and Implementation of a Mobile Internetworking," *Proc. of Winter USENIX*, Jan. 1993.

[17] Charles Perkins, "Providing Continuous Network Access to Mobile Hosts Using TCP/IP," *Joint European Networking Conference*, May 1993.

[18] F. Teraoka, Y. Yokote and M. Tokoro, "A Network Architecture Providing Host Migration Transparency," *Proc. ACM SIGCOMM Symposium on Communication, Architectures and Protocols*, 1991.

[19] C. Partridge, "Gigabit Networking," Addison Wesley, 1993.

[20] International Telecommunication Union Recommendation I.311 (03/93)

[21] J S Turner, "Design of a broadcast Packet switching network," *IEEE Transactions on Communications*, vol. 36, pp. 734-743, June 1988.

[22] Raif O Onvural, "Asynchronous Transfer Mode Networks : Performance Issues," Artech House, 1993.

[23] Martin de Prycker, "Asynchronous Transfer Mode, solution for broadband ISDN," Ellis Horwood 1991.

Appendix 1 : Two Dimensional Random Walk Model

We will first discuss the one dimensional random walk model as explained in [6], and then extend it to two dimensions.

Let the position of the user after t units of time be $x(t)$. Let the one dimension be the X axis. In the one dimension random walk model, every T units of time, the user tosses a coin, and based on the result the user either decides to go in the positive X direction or the negative X direction. For example, upon a head the user decides to take one step in the positive X direction, and upon a tail the user decides to take one step in the negative X direction. For the purpose of this discussion, we will assume that the step size if small enough so that a step in any other direction can be approximated by one of the four directions mentioned here.

The important parameters in the model are the time interval between two tosses T and the length of the step s. It is shown in [6] that for $t \gg T$, $x(t)$ is normally distributed with zero mean and variance αt as shown in the following equation.

$$f(x,t) = \frac{1}{\sqrt{2\pi\alpha t}} e^{-\frac{x^2}{2\alpha t}} \qquad (6)$$

where $\alpha = s^2/T$. It is assumed that the user starts from the origin. It is also assumed that the successive steps are independent of each other.

We can extend this analysis to two dimensional random walk model. Let the two dimensions be X and Y. Let positive X axis represent the east (E) direction, and the positive Y axis represent the north (N) direction. In such a model, the user tosses two coins every T seconds. Based on the resulting head-tail combination the user will decide to take a step in a specific direction. For example, a head-head results in the user taking a step in the north-east direction, a head-tail results in the user taking a step in the north-west direction, a tail-head results in the user taking a step in the south-east direction, and a tail-tail results in the user taking a step in the south-west direction.

We assume that the movement in the X-dimension is independent of the movement in the Y-dimension, and that the distribution functions are identical for both the dimensions. Thus, the joint density of the two dimensional random walk will be given as follows:

$$f(x,y,t) = \frac{1}{2\pi\alpha t} e^{-\frac{x^2+y^2}{2\alpha t}}$$

Let us assume circular cells of radius R units. At time t, the user will be in the same cell if $x(t) < R$, and $y(t) < R$. Therefore,

$$Prob(x(t) < R, y(t) < R) = \int_0^R \int_0^R f(x,y,t)\, dx\, dy$$

Converting into polar coordinates (r,θ) we get,

$$Prob(r(t) < R) = \int_0^{2\pi} \int_0^R f(r,\theta,t)\, J\, d\theta\, dr$$

where J is Jacobian [7], which is given as follows:

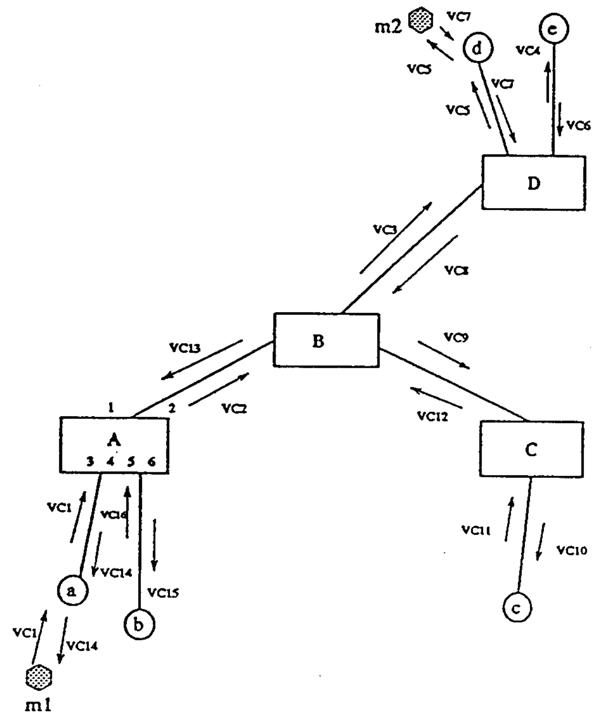
$$J = \frac{\partial(x,y)}{\partial(r,\theta)} = \begin{vmatrix} cos\theta & -rsin\theta \\ sin\theta & rcos\theta \end{vmatrix} = r$$

Replacing J, we get,

$$Prob(r(t) < R) = \int_0^R \frac{r}{\alpha t} e^{-\frac{r^2}{2\alpha t}}\, dr$$

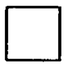
Therefore,

$$Prob(r(t) < R) = 1 - e^{-\frac{R^2}{2\alpha t}} \qquad (7)$$



VCN in	PORT in	VCN out	PORT out
1	3	2	1
16	5	2	1
13	2	14, 15	4, 6

Routing Information
For Switch A



VC - Virtual Circuit #

Figure 13: Dynamic Virtual Connection Tree

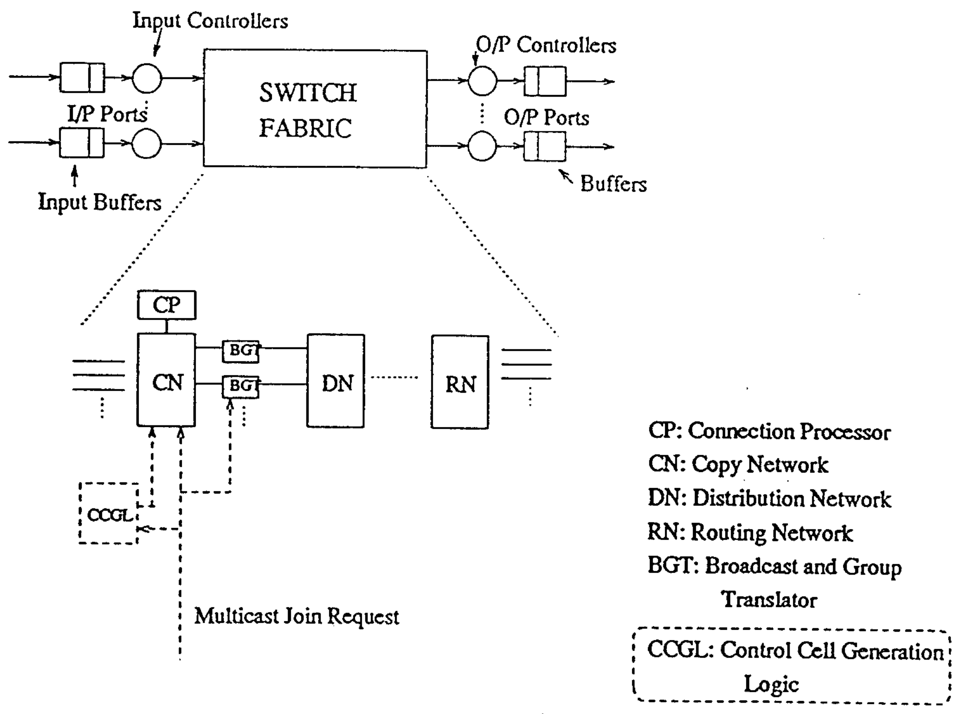


Figure 14: Switch Fabric modifications

IV. Algorithms for Loadbalancing

Processor Allocation in Hypercube Multicomputers: Fast and Efficient Strategies for Cubic and Noncubic Allocation

Debendra Das Sharma and Dhiraj K. Pradhan, *Fellow, IEEE*

***Abstract*—A new approach for dynamic processor allocation in hypercube multicomputers which supports a multi-user environment is proposed. A dynamic binary tree is used for processor allocation along with an array of free lists. Two algorithms are proposed based on this approach, capable of efficiently handling cubic as well as noncubic allocation. Time complexities for both allocation and deallocation are shown to be polynomial, a significant improvement over the existing exponential and even super-exponential algorithms. Unlike existing schemes, the proposed strategies are best-fit strategies within their search space. Simulation results indicate that the proposed strategies outperform the existing ones in terms of parameters such as average delay in honoring a request, average allocation time, average deallocation time, and memory overhead.**

***Index Terms*—Cubic allocation, deallocation, dynamic binary tree, fragmentation, hypercube, noncubic allocation.**

I. Introduction

In an MIMD hypercube supporting multiple users, an incoming task is allocated the required number of processors for execution. Upon completion of the task, these processors that were assigned to the task are released, for subsequent allocation; this process is known as "deallocation". Several processor allocation schemes have been proposed in the literature [1], [3], [5], [6], [7], [8], [13], [15], [16], [18], [24]. Most of these, including nCX, the host operating system of the nCUBE series [21], assume the number of processors as a power of 2. Many applications, though, do not necessarily require a complete subcube for execution [14], [23], [24], such as those requiring embedding of complete and incomplete binary tree, rectangular grid in hypercubes, solving a large number of nonlinear equations, etc.

Implementing a complete subcube allocation strategy has the drawback of allocating extra processors to the tasks to obtain a full subcube. Known as "internal fragmentation," this translates to both lower computation power as well as higher waiting times for subsequent tasks. Those strategies for noncubic allocation proposed in [15], [16], [24], have extremely high time complexities for allocation and deallocation (Table II), perhaps unsuitable for high dimensional hypercubes.

An efficient allocation scheme handles both cubic and noncubic allocation while exhibiting low time complexities for both allocation and deallocation, low memory overhead, high processor utilization, and low waiting times for incoming tasks. Two such schemes are presented here which possess the lowest time complexities and lowest memory overhead among existing schemes, while exhibiting superior performance.

A dynamic binary tree is used to represent the various subcubes in the proposed schemes. The dynamic binary tree maintains a very compact representation, which results in an extremely low memory overhead, shown in Section V. Incomplete subcubes are maintained explicitly in the dynamic binary tree. Representatives of the free incomplete subcubes are maintained in the form of an AVL tree associated with the highest dimension subcube in the incomplete subcube. Both proposed algorithms have polynomial time complexities ($O(n^2)$ and $O(n^3)$, respectively) for allocation and deallocation-orders of magnitude improvement over existing super-exponential algorithms, as depicted in Tables I and II. Although the proposed schemes lack complete subcube recognition capability, our incomplete subcube recognition capability is better than most of the existing schemes (Table II).

Simulation results demonstrate that the proposed schemes outperform the existing strategies for parameters like average waiting delay, variance in waiting delay, and average turnaround time for a wide range of workloads and dimensions of hypercube systems, while possessing the lowest overheads in allocation and deallocation times and amount of memory required. This effect is even more prominent as the dimension of the hypercube increases.

The paper is organized as follows: The following section presents pertinent preliminaries. A discussion of the existing methods is presented in Section III. Our approach is delineated in Section IV, including comparison against existing schemes. Section V provides simulation results, comparing the performance of our strategy against certain existing ones. The conclusion is contained in Section VI.

II. Preliminaries

We consider an n-dimensional hypercube where the individual nodes or subcubes are represented by an n-bit string of ternary symbols from $\Sigma = \{0, 1, x\}$, where x denotes a "don't care." For example, in a 2-dimensional hypercube $1x$ denotes the nodes 10 and 11, and xx denotes all the four nodes.

Manuscript received July 9, 1993; revised July 5, 1994.
D. Das Sharma is with the General Systems Lab, Hewlett-Packard Company, Roseville, CA 95747-5596; e-mail: dsharma@hprpcd.rose.hp.com.
D.K. Pradhan is with the Department of Computer Science, Texas A&M University, College Station, TX 77843-3112; e-mail: pradhan@cs.tamu.edu.
IEEECS Log Number D95040.

DEFINITION 1. The **Hamming Distance** [15] between two subcubes $a = a_1a_2 \ldots a_n$ and $b = b_1b_2 \ldots b_n$; where $a_i \in \Sigma$ and $b_i \in \Sigma$ for all $i \in [1, n]$; can be defined as $H(a, b) = \Sigma_{i=1}^{n} h(a_i, b_i)$, where $h(a_i, b_i) = 1$ if $a_i \neq b_i$ and $a_i, b_i \in \{0, 1\}$, and 0 otherwise. For example, $H(00x, 1xx) = 1$ and $H(1x0, x0x) = 0$.

DEFINITION 2. The **Exact Distance** [15] between the two subcubes a and b above, can be defined as $E(a, b) = \Sigma_{i=1}^{n} e(a_i, b_i)$ where $e(a_i, b_i) = 0$ if $a_i = b_i$ and 1 otherwise. For example, $E(0x1, 10x) = 3$ whereas $H(0x1, 10x) = 1$.

DEFINITION 3. An **Incomplete Subcube** (ISC) S can be defined as follows:

1) It consists of a group of disjoint subcubes $\{S_1, S_2, \ldots, S_m\}$, $(1 \leq m \leq n)$, with dimensions $d_1, d_2, \ldots, d_m$, respectively, $(d_1 > d_2 > \ldots > d_m)$[1].
2) $H(S_i, S_j) = 1$ for all $1 \leq i, j \leq m$, $i \neq j$.
3) $E(S_i, S_j) = d_i - d_j + 1$ for all $1 \leq i \leq j \leq m$.

d_1 is the **dimension** and $d = \Sigma_{i=0}^{m} 2^{d_i}$ is the **size** of the ISC S. S_1 is called the **head** of the ISC S.

EXAMPLE 1. The subcubes $\{1xxx, 00xx, 010x\}$ form a 11-node ISC (i.e., size = 11) of dimension 3 and $1xxx$ is the head. But subcubes $\{00xx, x100\}$ do not form an ISC as the exact distance between them is 4 instead of $2 - 1 + 1 = 2$; although the hamming distance is 1.

Essentially, an ISC consists of subcubes such that the hamming distance between any two subcubes is 1 and any higher dimensional subcube has xs in the same positions that any other lower dimensional subcube has. This ensures that all the processors in an ISC lie within the next higher dimensional subcube, which minimizes the extent to which tasks can have overlapping links.

DEFINITION 4. A **binary representation** of a hypercube is a dynamic binary tree, where nodes in level i denote a subcube of dimension $n - i$. Any node that is allocated to one task or a free node does not have any descendants. The binary representation of nodes is as shown in Fig. 1. For example, the root node (level 0) represents the entire hypercube, its left child denotes the subcube $0xxx$, right child denotes the subcube $1xxx$, and so on. Node $1xxx$ is allocated to a task (along with node 0000) and hence does not have any descendants. Node $01xx$ is free, and does not have any descendants. This helps us maintain a space-efficient representation of the hypercube.

DEFINITION 5. A **Sibling generated Incomplete SubCube** (SISC) is an incomplete subcube consisting of the subcubes $\{S_1, S_2, \ldots, S_m\}$ $(1 \leq m \leq n)$, arranged in the decreasing order of dimensions, such that the sibling of S_i is the common ancestor of all the subcubes $\{S_{i+1}, S_{i+2}, \ldots, S_m\}$, for all $i \in [1, m - 1]$; in the binary tree representation. The *dimension* of the SISC is the dimension of S_1 and the total number of processors define its *size*.

1. For the rest of this paper, it will be assumed that the subcubes are arranged in the decreasing order of dimensions, unless otherwise mentioned.

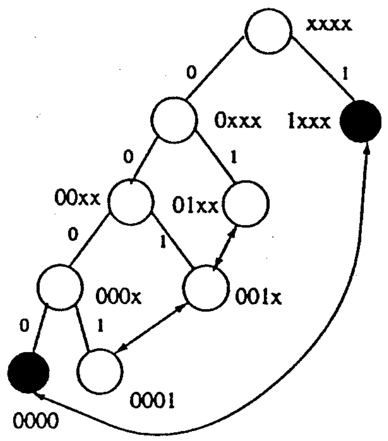


Fig. 1. Dynamic binary tree representation.

EXAMPLE 2. (Fig. 2) Nodes $\{1xxxxxx, 00xxxxx, 0110xxx, 01111xx, 011100x, 0111011\}$ form a SISC (say S), as $0xxxxxx$, the sibling of subcube $1xxxxxx$, is the ancestor of all the lower dimensional subcubes ($00xxxxx$, $0110xxx$, etc.); $01xxxxx$, the sibling of the node $00xxxxx$, is the ancestor of all the lower dimensional subcubes ($0110xxx$, $01111xx$, etc.), and so on. The dimension of this SISC is 6 and its size is 64 + 32 + 8 + 4 + 2 + 1 = 111. Similarly, nodes $\{0101xxx, 01000xx, 010010x\}$ form a SISC (say S') in the tree.

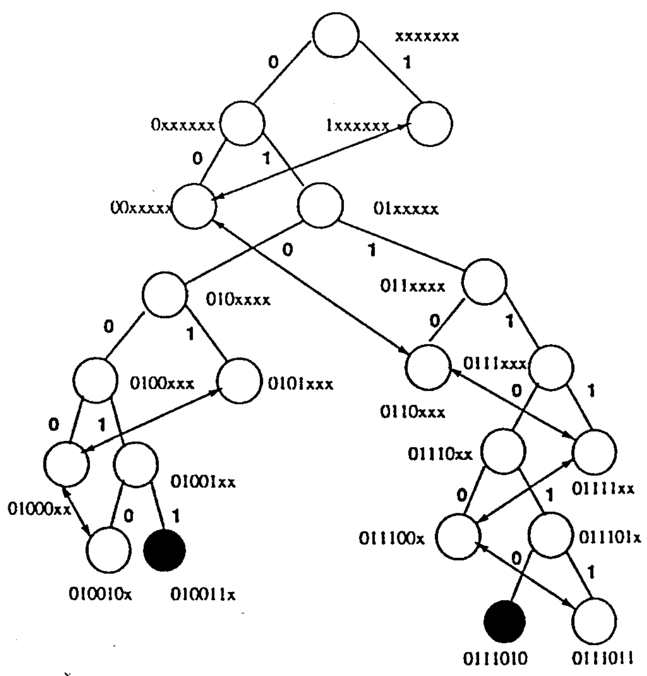


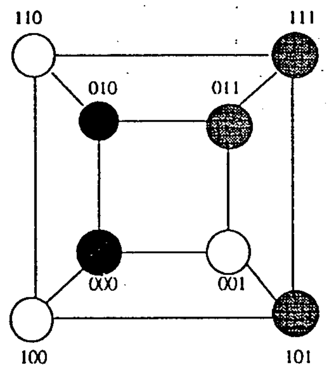
Fig. 2. Incomplete subcube representation.

THEOREM 1. *SISC is an ISC.*

The proof appears in the appendix.

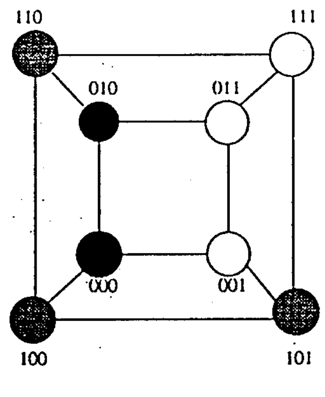
DEFINITION 6. **Maximal Set of Incomplete Subcubes** (MSIS) is a set of free, disjoint ISCs that is greater than or equal to all other sets of free disjoint ISCs of the same set of free nodes. For example, in Fig. 2, ISCs {1*xxxxxx*, 00*xxxxx*, 0101*xxx*, 01000*xx*, 010010*x*} (size 110) and {0110*xxx*, 01111*xx*, 011100*x*, 0111011} (size 15) do not form an MSIS because *S* and *S'* of the previous example form an MSIS with sizes 111 and 14, respectively.

DEFINITION 7. **Physical Fragmentation** occurs when a sufficient number of free processors can not form an incomplete subcube of the required size. Fig. 3a illustrates an example where the three free nodes do not form an incomplete subcube of size 3. Throughout this paper a shaded node will indicate an allocated subcube; an unshaded one, a free subcube. The physical fragmentation problem is similar to that of memory fragmentation and may result from the sequence of incoming and outgoing tasks or simply a "bad" allocation (Example 3).



Fragmentation due to a bad allocation

(a)



No fragmentation

(b)

Fig. 3. Allocation in a 3-dimensional hypercube.

EXAMPLE 3. Consider the 3-dimensional hypercube shown in Fig. 3. If an incoming request requiring 3 nodes is allocated {1*x*1, 011} (Fig. 3a) instead of {10*x*, 110} (Fig. 3b), a subsequent request for a 3-node incomplete subcube cannot be allocated. Similarly, if an incoming request of dimension 1 is allocated the subcube 10*x* instead of 1*x*0, a subsequent request for a 2-dimensional subcube has to wait.

DEFINITION 8. **Virtual Fragmentation** occurs when an allocation policy fails to recognize an existing incomplete subcube and causes an incoming task to wait. For example, if an allocation policy fails to reconize the 3-node free ISC {0*x*1, 111} in Fig. 3b, an incoming request requiring three processors has to wait.

III. EXISTING ALLOCATION SCHEMES

The existing allocation strategies may be broadly classified into two categories.

- *Bit mapping strategies* such as Buddy [18], Gray Code (GC) [3], [5], Modified Buddy [1], and Tree Collapsing (TC) [6].
- *List type Strategies* such as the Maximal Set of Subcubes (MSS) [13], Free List (FL) [15], [16], and Prime-Cube Graph (PC-Graph) [24].

The bit mapping strategies maintain 2^n bits, each representing the (un)availability of the corresponding processor. These bits are searched to determine the availability of a subcube, and the first available subcube is allocated. Noncubic allocation may be handled by searching for and allocating only the required number of processors instead of a subcube. The problems associated with these schemes are the physical fragmentation due to their first-fit nature and the virtual fragmentation due to their incomplete recognition capability (both for cubic and noncubic case, as illustrated in Tables I and II). These limitations may result in degraded performance in terms of the average waiting delay [12]. In addition, the bit-mapping strategies have exponential time complexities for allocation and deallocation (Tables I and II). Thus, they may not be suitable for processor allocation in hypercubes of large sizes.

The list type strategies maintain a list of the free subcubes in the system for subcube allocation. They have complete subcube recognition capability and are best-fit strategies for subcube allocation. However, some of them do not possess complete ISC recognition capability (Table II). The use of heuristics for both allocation and deallocation may potentially cause both physical and virtual fragmentation, even for cubic allocation, as illustrated in [7], [12]. The problems are worse for noncubic allocation. Recognizing ISCs in the list type strategies is non-trivial, as the relationship among free subcubes in the various lists has to be examined. This may involve super-exponential time complexity as there can be $O(2^n)$ free subcubes and $O(n)$ free subcubes of distinct dimensions can form an ISC. This time complexity may further increase if we try to allocate the "best" ISC in order to reduce physical fragmentation and reducing common links between tasks [12]. This is probably the reason the list type strategies are first-fit for noncubic allocation (Table II). This results in degraded performance, as illustrated in Section V. A detailed comparison of the various strategies appears in [12].

We define the goal of an allocation algorithm to be able to maintain the MSIS after every allocation and deallocation. Maintaining the MSIS after every allocation and deallocation will involve the undesirable effect of super-exponential time complexities due to the inherent nature of the problem. Thus,

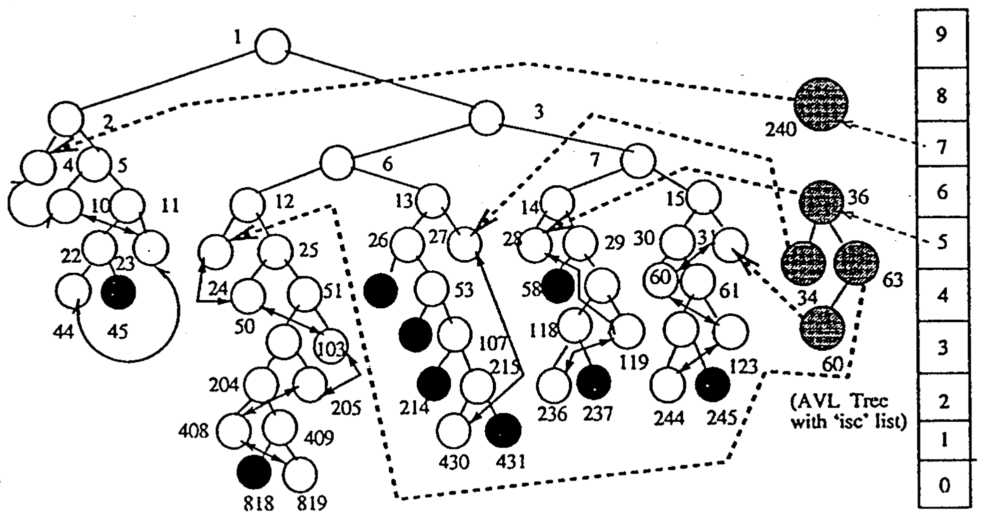


Fig. 4. Representation of ISCs in the *isc* lists by lightly shaded nodes arranged in the form of an AVL tree.

we use a strategy that recognizes ISCs of type SISC only and maintain the MSIS of ISCs of type SISC only. The allocation and deallocation time complexities of the proposed strategies is polynomial. Because they are best-fit strategies, the proposed allocation schemes reduce physical fragmentation significantly. They exhaustively coalesce a released subcube if it is recognizable by the strategy. In addition, the proposed schemes are capable of recognizing the adjacency of up to n subcubes in the tree, unlike free list and prime cube; the only two noncubic allocation strategies in the literature. We also do not incur the penalty of higher job turn-around time due to reduced bisection bandwidth or shared links, unlike the prime-cube approach [12], as we allocate only ISCs, contained within the next higher dimensional subcube, instead of any arbitrary noncubic structure. Although the proposed strategies do not have complete recognition ability for cubic as well as noncubic allocations, simulation results indicate that the strategies exhibit much better performance than the existing strategies. Thus, by limiting our subcube (and hence the ISC) recognition capability we obtain polynomial time algorithms whose performance and even the ISC recognition capability exceed that of the existing strategies; as illustrated later.

IV. The Proposed Strategy

In the proposed strategy, a dynamic binary tree along with an array of free lists is used for processor allocation. The nodes in the tree represent various subcubes. Free subcubes may join to form an ISC of type SISC, as explained in Section II. For the rest of this section, "ISC" refers to SISC only and "MSIS" refers to MSIS of type SISC only.

Incomplete subcubes are represented in the algorithm by a bidirectional link between two adjacent subcubes; the subcubes being arranged in the decreasing order of their dimensions. The highest dimensional subcube is the head of the ISC. Each free ISC is represented by a separate type of node (shown as lightly shaded nodes in Fig. 4) which stores the number of processors and the address of the head of the ISC it represents. Representatives of all the free ISCs are kept in an array of lists called "*isc*," which has $n + 1$ entries in it. Representatives of all the free ISCs of dimension i are arranged as a height-balanced AVL tree [19], the key being the number of processors in the ISC each node represents. *isc*[i] points to the root of the AVL tree associated with the ISCs of dimension i. The AVL tree is a height-balanced binary tree, where the difference of the depths of the right and left subtrees of any node is at most one [19]. The AVL tree ensures that search, insert and delete operations can be done in $O(log(k))$ time [19], where k is the number of ISCs of dimension i ($k = O(2^{n-i})$). This, in turn, ensures polynomial time complexities of the proposed algorithms.

Example 4. The dynamic binary tree represented in Fig. 4 has five ISCs. The ISC formed by the nodes {4, 10, 23, 44} has 240 processors and is of dimension 7. It is the only ISC of dimension 7. Hence, *isc*[7] points to the representative node of the ISC (the only node in that AVL tree). The rest of the ISCs in the dynamic binary tree are all of dimension 5 with 34, 36, 60, and 63 processors in them. Thus, the AVL tree associated with the dimension 5 has four entries in it, as illustrated in the figure. Each representative node contains the number of processors, and the address of the head of the ISC it represents (indicated by dotted lines in the figure) along with the necessary information to maintain the height balanced AVL tree. The rest of the entries in *isc* are set to *NULL*.

A. Noncubic Allocation

In this subsection, we present two algorithms for noncubic allocation that work efficiently for cubic allocation as well. The algorithms proposed in this subsection will use the data structure described above.

A.1. Algorithm 1

In this approach, each node (N) in the dynamic binary tree maintains a pointer to the head of the largest ISC (denoted as "*iscptr*") beneath N (all the subcubes in this largest ISC are descendants of N) along with the number of processors in the ISC (denoted as "*iscnodes*"). For instance, in Fig. 4, the *iscptr* entry of node 3 will point to node 24, the head of an ISC with 63 processors. Thus, its *iscnodes* entry is 63. A node allocated to one task, a free node and a nonleaf node, all of whose descendants are allocated, will have the *iscptr* set to NULL and *iscnodes* set to 0.

During allocation, the ISC S, with the minimum number of processors that can satisfy the request, is chosen for allocation. The representative of this ISC is removed from its corresponding AVL tree (and hence the *isc* list). Subcubes of the required dimensions are allocated to the incoming task (I_j) by using this ISC. The free subcubes of S, that were not used for allocation to I_j, are released to form one (or two) ISC(s) and added to the corresponding *isc* list(s). It should be noted that every insertion/deletion of an ISC to/from an *isc* list (AVL tree) is accompanied by updating the *iscptr* (and *iscnodes*) entry of the ancestor(s) of the head of the ISC.

A formal description of the allocation algorithm is described below followed by examples describing the various steps of the allocation procedure. In this algorithm *allocatedlist* maintains the list of subcubes to be allocated to the task after execution of the algorithm, *newlist* maintains the list of higher dimensional subcubes that are not required for the current allocation in progress, *brokenlist* maintains the list of free subcubes generated after allocating subcubes to the request I_j by fragmenting a higher dimensional subcube.

A.1.a. Allocation

Step 1. Form the dimensions $D_1, D_2, \ldots, D_m$ of the subcubes required to satisfy the task I_j requesting D processors; arranged in the decreasing order.

Step 2. Search the *isc* lists from *isc*[D_1] onwards and choose the ISC $S = \{S_1, S_2, \ldots, S_t\}$ with the minimum number of processors that can satisfy request I_j (let the dimensions of subcubes in S be $d_1, d_2, \ldots, d_t$, respectively). If no such ISC is found, keep the request in the waiting queue and skip the remaining steps.

[The allocation process tries to allocate the subcubes of dimensions $D_1, D_2, \ldots, D_m$, respectively, (in that order) from the free subcubes $S_1, S_2, \ldots, S_t$ (in that order). S_i denotes the subcube in S that is currently being considered for allocating subcubes of dimensions $D_j, D_{j+1}, \ldots, D_m$.]

Step 3. $i = j = 1$. *allocatedlist* = *newlist* = *brokenlist* = *NULL*. (Initialization).

[d_i is the highest dimension subcube in S that may be allocated to I_j. D_j is the highest dimension subcube in I_j that is being considered for allocation. The following steps illustrate the various scenarios that may arise based on the relationship between d_i and $D_j, D_{j+1}, \ldots, D_m$.]

Step 4. If $d_i > D_j$ and the available subcubes $\{S_{i+1}, S_{i+2}, \ldots, S_t\}$ of S can satisfy the request for the subcubes of dimensions $D_j, D_{j+1}, \ldots, D_m$ yet to be allocated:

Remove S_i from S and append it to *newlist*. $i = i + 1$. Go To Step 4. (Example 5)

(In this Step, subcube S_i need not be allocated, since the lower dimensional subcubes of S can satisfy the request.)

Step 5. If $d_i = D_j$: (*i.e., S_i has to be allocated*)

1) Append S_i to *allocatedlist* after removing it from S (Example 5). $j = j + 1$.
2) If $j = m + 1$ (*all subcubes of the required dimensions allocated*):
   - *list* = $\{S_{i+1}, \ldots, S_t\}$ and Go To Step 7. (Step 7 releases the free subcubes in *list* and *newlist* to the system. These subcubes were not used in the allocation process.)
   - Allocate the subcubes in *allocatedlist* to I_j. Skip the remaining steps.
3) If the sibling S_i' of S_i has enough processors in the maximal ISC beneath it (denoted as $\{\overline{S}_1, \overline{S}_2, \ldots, \overline{S}_l\}$) to accommodate subcubes of the remaining dimensions $D_{j+1}, D_{j+2}, \ldots, D_m$, do the following: (Example 6)

(trying to see if ISCs of lower sizes can be used instead of the remaining subcubes in S)

   a) Remove the ISC headed by $\overline{S}_1$ from the *isc* list and update its ancestors.
   b) *list* = $\{S_{i+1}, S_{i+2}, \ldots, S_t\}$. $S = \{\overline{S}_1, \overline{S}_2, \ldots, \overline{S}_l\}$. $i = 1$. Go to Step 7. (Example 7)

(Step 7 tries to form maximal ISC(s) out of unused subcubes of previous ISC S.)

Else: $i = i+1$.

4) Go to Step 4.

Step 6. If $d_i > D_j$: (*Now subcube S_i has to be used to satisfy the remaining subcube requests of dimensions $D_j, D_{j+1}, \ldots, D_m$ by fragmenting S_i (Example 8).*)

1) Form the two children of S_i in the dynamic binary tree (of 1 dimension less).
2) If the dimension of the children = D_j:
   a) Allocate the right child to I_j by appending it to *allocatedlist*.
   b) S_i = left child of S_i.
   c) $j = j + 1$.

Else:

[Since the dimension of node S_i's children is greater than D_j (the highest dimensional subcube yet to be allocated); one of S_i's children needs to be further fragmented to allocate a subcube of dimension D_j. The right child of S_i would be used to allocate the remaining requests and the left child would be eventually released.]

   a) Append the left child to *brokenlist* (to be released later).
   b) S_i = right child of S_i (to be used for allocation).
3) If $j \le m$ Go To Step 6.1. (Continue allocation as more subcube(s) are yet to be allocated.)
4) If the number of processors in *brokenlist* is greater than that in $\{S_{i+1}, S_{i+2}, \ldots\}$:

(The free subcubes remaining after fragmenting the original node S_i in the ISC S have more processors than the remaining free subcubes in S (i.e., $S_{i+1}, S_{i+2}, \ldots, S_t$). Hence, *brokenlist* is chosen to combine with higher dimensional free subcubes instead of the remaining free subcubes in S.)

- Insert $\{S_{i+1}, S_{i+2}, ..., S_t\}$ to the appropriate *isc* list and update the ancestors (i.e., their *iscptr* and *iscnodes* entries) of S_{i+1}.
- *list* = *brokenlist* (to combine with higher dimensional subcubes, if possible).

Else:

[Since the number of processors in the ISC S ($\{S_{i+1}, S_{i+2}, ..., S_t\}$) is greater than those in *brokenlist*, the ISC S is chosen to combine with higher dimensional free subcubes, if possible.]

- Insert *brokenlist* to the appropriate *isc* list and update the ancestors of the highest dimensional subcube in *brokenlist*.
- *list* = $\{S_{i+1}, S_{i+2}, ...\}$ (to combine with higher dimensional subcubes, if possible)

5) Go to Step 7.
6) Allocate the subcubes in *allocatedlist* to the task I_j. Skip the remaining steps.

(Step 7 is used by Steps 5 and 6 only. It forms maximal ISCs out of the free subcubes in "*list*" and "*newlist*" and returns to the point from which it was invoked.)

Step 7.

1) *If newlist is empty:* [*No more free (higher dimensional) subcubes with which the (lower dimensional) subcubes in "list" can combine to form a higher dimensional ISC*]
   - Insert the subcubes in *list* to the appropriate *isc* list.
   - Update the ancestors of its head.
   - RETURN.
2) Let L be the lowest dimension subcube of *newlist* and L_s be its sibling.
3) Remove L from *newlist*.
4) If the maximal ISC beneath L_s (say I_s) has more processors than *list*:

(In this case, the maximal ISC beneath L_s (i.e., I_s) should combine with L instead of the ISC in *list*, as that would yield a maximal ISC. Hence, I_s is removed from the "*isc*" list and combined with L, whereas the ISC in *list* is added to the appropriate "*isc*" list.)

- Insert the subcubes in *list* to the appropriate *isc* list and update the ancestors of its head.
- Remove I_s from the *isc* list and update the ancestors of its head.
- *list* = $\{L\}$ append $\{I_s\}$.

Else: Add L to the head of *list* (Example 7).

5) Go to Step 7.1.

EXAMPLE 5. Consider the scenario of Fig. 5a. Suppose we have a request for 149 processors (dimensions 7, 4, 2, 0) (Step 1). The ISC headed by node no. 2 (S = {2, 7, 25, 48, 99}) is selected from *isc*[8] as *isc*[7] is empty (Step 2). The subcube no. 2 of dimension 8 in S need not be allocated as the rest of the subcubes in S can satisfy the request (Step 4). Following Step 5.1, Node 7, however, needs to be allocated (kept in *allocatedlist*) whereas node 2 will be released later (goes to *newlist*).

EXAMPLE 6. (Continuing from Example 5) The sibling of the allocated node 7 (node 6) is checked to see if the maximal ISC beneath it (the ISC {55, 109, 217} with 28 processors) can satisfy the remaining dimensions 4, 2, and 0. Since, the ISC headed by 55 can accommodate the remaining subcubes, we use that instead of the subcubes {25, 48, 99} in an attempt to preserve higher ISCs (Fig. 5b). (Now *list* = {25, 48, 99} and S = {55, 109, 217}. This constitutes Step 5.3.) However, the choice of ISC {53, 210, 847} would have maintained the MSIS after allocation. This drawback arises as nodes do not store all the ISCs beneath them to make the "best" choice. The second algorithm overcomes this shortcoming by maintaining all the ISCs beneath a node in the form of an AVL tree instead of maintaining the maximal ISC only.

It should be noted that the maximal ISC beneath the sibling of an allocated subcube will have fewer processors than the remaining subcubes in S, as the ISC with more processors always combines with higher dimensional free subcubes to form a larger ISC.

EXAMPLE 7. (Continued from Example 6) (Fig. 5b) After the ISC headed by 55 is chosen for allocation, the free subcubes of the current ISC (headed by 2) are to be deallocated. These free subcubes belong to two categories: those higher dimensional subcubes of S that were not used in allocation (kept in *newlist* during Step 4; here *newlist* = {2}) and those lower dimensional subcubes (stored in *list*) that were not used during allocation either because the allocation process was complete (Step 5.2) or because a smaller ISC was available (Step 5.3) (here *list* = {25, 48, 99}). After removing the ISC headed by 55 from *isc*[4], the sibling of node 2 (which is the lowest dimension subcube in *newlist*) is checked for its maximal ISC. In this case, the current ISC (*list*) has more numbers of processors than the maximal ISC beneath 3 (headed by 53 with 21 processors). Thus, we add node 2 to the remaining free subcubes of the current ISC and add {2, 25, 48, 99} in the free list of *isc*[8] and update the ancestors of 2 (Step 7 followed by Step 5.3). If the ISC headed by 53 had more processors than *list*, it would have been removed from its *isc* list and combined with node 2 to form a new ISC and *list* would have been added to the corresponding *isc* list (part of Step 7.4).

EXAMPLE 8. (Continued from Example 7) The ISC S = {55, 109, 217} is now used to allocate the remaining subcubes of dimensions 4, 2, and 0 to I_j. Node 55 of dimension 4 will be allocated (Fig. 5c). Node 109 of dimension 3 needs to be fragmented (Step 6) to allocate the remaining 5 processors to I_j (Fig. 5d). Nodes 219 and 875 generated from node 109 are allocated to I_j, whereas the unused nodes 436 and 874 need to be released and are stored in *brokenlist* (*brokenlist* = {436, 874}). Since the unallocated node 217 from S has more processors than *brokenlist* (Step 6.4), the latter simply joins *isc* [1] whereas the former is chosen to combine with higher dimensional subcubes, if possible (Step 7). However, in this case, node 217 does not have enough processors to join node 2 and is kept in *isc*[2].

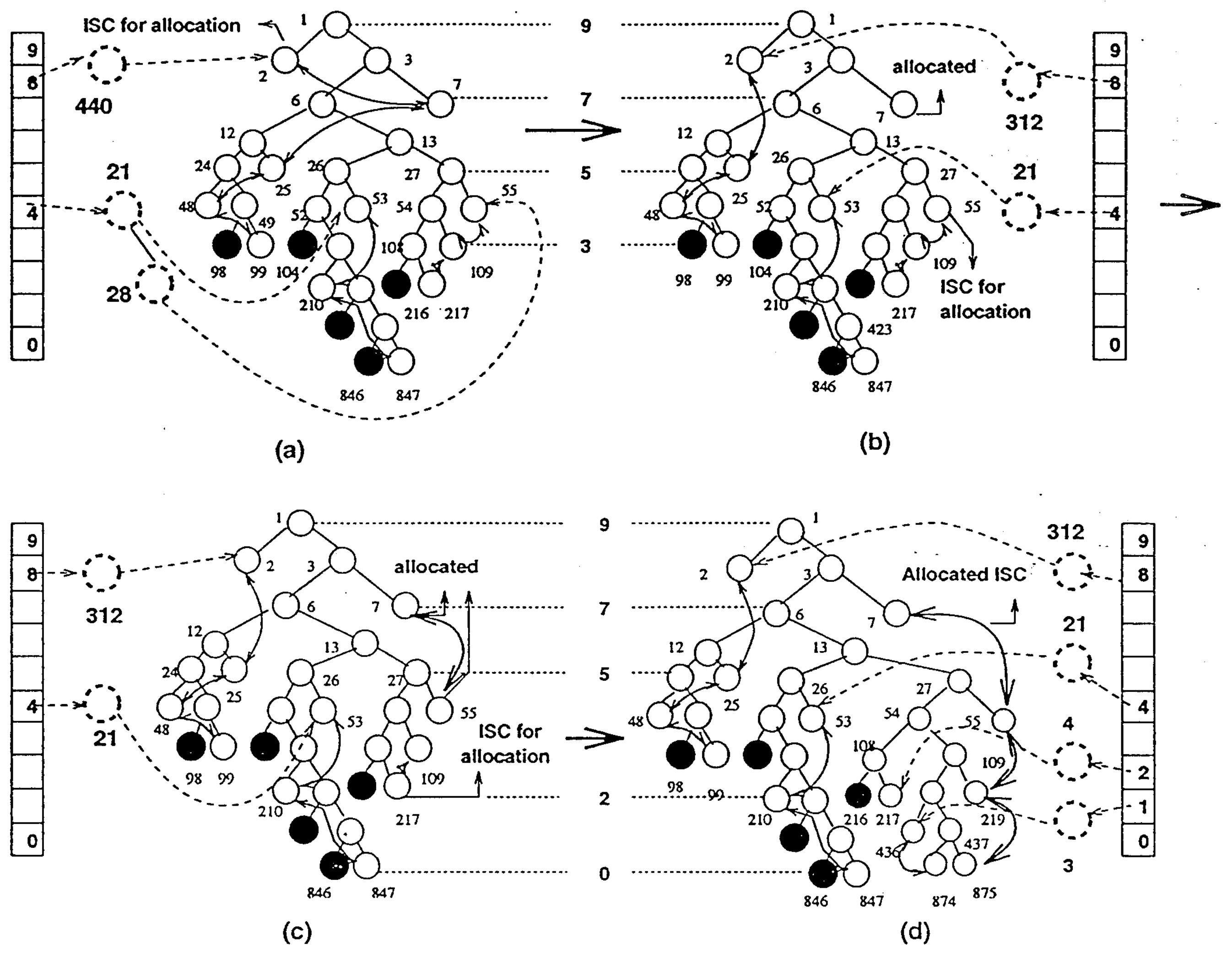


Fig. 5. Allocating an ISC of size 149 in a 9D hypercube.

A.1.b. Deallocation

During the deallocation process, the proposed strategy tries to combine the released subcubes with the existing free subcubes to form the MSIS. Theorem 2 proves that the deallocation procedure always maintains the MSIS (though the same can not be said about the allocation process). The deallocation process consists of two phases. In the first phase, free sibling subcubes are combined and the parent is marked free. This process continues until the sibling of the free subcube is not free or the entire hypercube is found to be free (Example 9). In the second phase, free subcubes of different dimensions are grouped to identify free ISCs in the system. A formal description of the algorithm is described below followed by examples.

Step 1. Let $L = \{S_1, S_2, ..., S_m\}$ be the subcubes of the ISC (to be deallocated) arranged in the *increasing* order of their dimensions $d_1, d_2, ..., d_m$, respectively.

(Steps 2-4 try to combine free sibling nodes to form higher dimensional free subcubes as much as possible.)

Step 2. $i = 1$. Remove S_1 from L.

Step 3. If the sibling S_i^s of S_i is free:
(i.e., S_i^s has no subcubes beneath it in its ISC now as S_i is being released as a free node (and hence can not have any free descendant nodes with whom S_i^s can combine to form an ISC).)

- If S_i^s is the head of the ISC: Remove the ISC from the *isc* list and update the ancestors of S_i^s.
- Combine the two siblings S_i^s and S_i by removing them from the dynamic binary tree and mark their parent as S_i. Go To Step 3.

Step 4. If $S_i^s = S_{i+1}$ (*i.e., the sibling is itself among the released subcubes.*)

- Remove S_{i+1} from L. Eliminate S_i and S_{i+1} from the tree and mark their parent as S_{i+1}.
- $i = i + 1$. Go To Step 3.

(Combining free sibling nodes to form the free parent node ends here. Now we have to form maximal ISCs out of the released subcubes by traversing up the tree. We start to form the ISC(s) by starting from the lower dimensional subcubes. S_i is the lowest dimensional free subcube. S denotes the ancestor of

S_i whose sibling S^s is examined to form the maximal ISCs.)

Step 5. If S_i is the root of the dynamic binary tree (i.e., the entire hypercube is free): insert S_i in *isc*[CUBESIZE] and skip the remaining steps.

Step 6. $S = S_i$. S_i^s = Sibling of S_i; I_i^s = maximal ISC under S_i^s. *list* $= \{S_i\}$. $L = L - \{S\}$.

(*S_i being the lowest dimensional free subcube in L, I_i^s denotes a lower dimensional ISC that can be readily appended to L to form a higher ISC. "list" maintains the free subcubes that will form an ISC.*)

Step 7. S = Parent of S. If S is not root of the tree: S^s = Sibling of S.

Step 8. If S is the root of the dynamic binary tree: (search over)

1) If I_i^s is not NULL: Remove I_i^s from the corresponding *isc* list; Update ancestors of the head of I_i^s and *append* I_i^s to *list*.
2) Insert *list* in the appropriate *isc* list and update the ancestors of its head.
3) Skip the remaining steps.

(Step 9 tries to combine the sibling S^s of the ancestor S of the released subcube S_i.)

Step 9. If S^s is free, do the following steps:

1) If S^s has more processors beneath it in its ISC than those in *list* and I_i^s combined (i.e., it will yield an ISC of higher size if we choose the ISC to which S^s belongs; instead of the current *list*):
   - If I_i^s is not NULL remove the ISC I_i^s from the *isc* list and update the ancestors of its head. Append I_i^s to *list*. I_i^s = NULL.
   - Insert the ISC formed by *list* in the appropriate *isc* list and update the ancestors of the head of *list*.
   - If L is empty skip the remaining steps. (No higher ISCs possible.)
   - If S^s is the head of an ISC: Remove the ISC headed by S^s from the *isc* list and update the ancestors of S^s and name them as *list*. Else: Remove the subcubes from S^s onwards from the ISC and name them as *list*.
   - $S = S^s$. *list* = S^s append *list*.

   Go To Step 7.
2) (S^s has fewer processors beneath it than *list* and I_i^s. Hence, the ISC to which S^s belongs is altered. The subcubes of dimensions lower than S^s are discarded and subcubes in *list* is combined with S^s instead. The discarded subcubes form a separate ISC.)
   - If S^s is head of an ISC: Remove the ISC from the corresponding *isc* list and update the ancestors of S^s.
   - Remove the subcubes beneath S^s in the ISC and form a new ISC (call it N).
   - If the subcubes in N are descendants of S_i^s and $S_i \in$ *list*: Append N to *list*. I_i^s = NULL.

   Else If I_i^s is not NULL:
     - Remove I_i^s from the *isc* list and update the ancestors of its head.
     - Append I_i^s to *list*. Set I_i^s = NULL.
     - Insert N to the appropriate *isc*. Update ancestors of its head.
   - *list* = S^s append *list*. Go To Step 7.

Step 10. If $S^s = S_{i+1}$: Remove S_{i+1} from L; Add S_{i+1} to the head of *list*, $i = i + 1$ and Go To Step 7. (The sibling node S^s turned out to be a higher dimensional released subcube.)

(Since S^s is not free we check if any of its highest descendant ISC (*iscptr*), if any, has more processors than what we have accumulated so far from the released nodes. If so, this ISC will combine with the higher dimensional released node to form a higher ISC.)

Step 11. If S^s is not free (partially allocated) and has more processors in the maximal ISC beneath it than those in *list* and I_i^s combined do the following steps. (Since the maximal ISC beneath S^s has more processors than our current *list* we choose the former to combine with higher dimensional subcubes.)

1) If I_i^s is not NULL: Remove I_i^s from the corresponding *isc* list and update the ancestors of its head. Append I_i^s to *list*.
2) Insert the ISC *list* in the appropriate *isc* list. Update the ancestors of its head.
3) If S = NULL: Skip the remaining steps.
4) Remove the highest ISC beneath S^s from the *isc* list; update its ancestors. *list* = this new ISC.

Step 12. Go to Step 7.

The following examples illustrate the deallocation procedure. More detailed examples of this procedure appear in [12]. The deallocated nodes are indicated in the figures by an unshaded circle that is crossed. The newly formed ISC is indicated by solid dotted bidirectional arrows whereas subcubes yet to be considered for deallocation are connected by a solid bidirectional arrow.

EXAMPLE 9. Let us consider the ISC L = {2, 12, 208, 418, 1,672} that is being released (Fig. 6a). The first phase tries to collapse the tree by removing free siblings. We start with node 1,672 (Step 2) which combines with 1,673. The two nodes are removed from the tree and their parent node 836 is marked free (Step 3), which combines with node 837, making their parent 418 free. Node 418 combines with a released node 419 (removed from L); their parent 209 becomes free (Step 4). Node 209 combines with its sibling 208, another released node, and their parent (node 104) is marked free. The procedure terminates at node 52 since its sibling (node 53) is not free. All the descendants of node 52 were continuously removed at each step of this collapsing phase.

It should be noted that in the deallocation process, the entry for the free ISC with whom the released subcubes interact, is not changed, until the head of the ISC is involved. The deallocation process is guaranteed to continue till the head of the ISC once any of its members are changed in the deallocation process (Step 9.2). This helps us maintain a low time overhead by updating the *isc* entry and ancestors of the head of each affected ISC exactly once.

EXAMPLE 10. (Continued from Example 9.) After the first phase, we have the scenario of Fig. 6b. Now L = {2, 12}, *list* = {52}, I_i^s = NULL (Step 6). The free nodes will now be combined to form ISC(s). Node 52 combines with node 27 (*list* = {27, 52}, Fig. 6c). This process continues to include

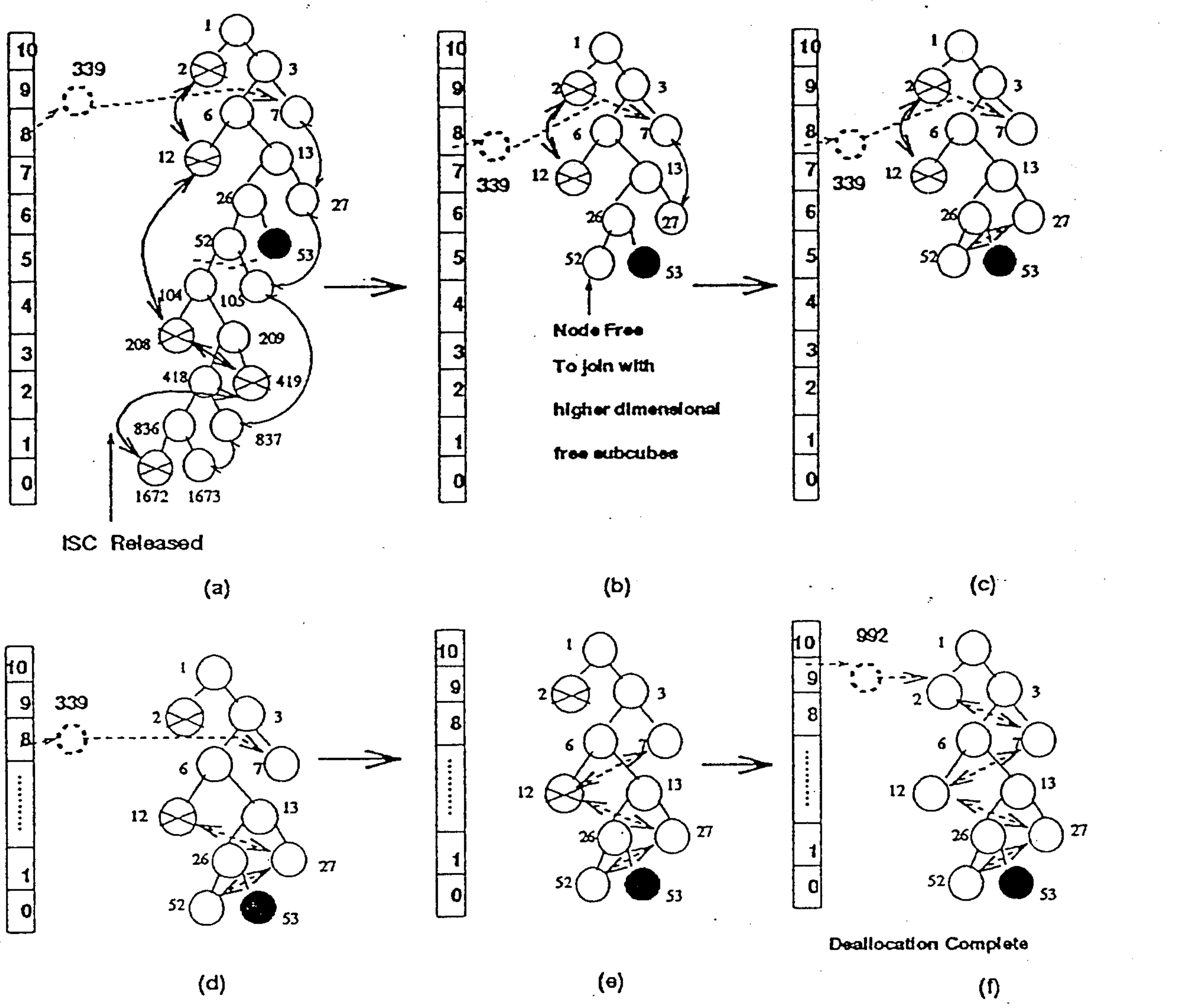


Fig. 6. Deallocation in a 10D hypercube.

node 12 (Fig. 6d) and node 7 (Fig. 6e). The inclusion of node 7 indicates that the entry for the old ISC it was heading has to be removed from *isc*[8] (Fig. 6e) and all the ancestors of node 7 have to be updated. The new ISC (*list*) is {7, 12, 27, 52}. The inclusion of all these nodes was due to the repeated execution of Step 9.2. The search process terminates after including node 2 to form the ISC (Step 10) and inserting an entry for 992 processors for the newly formed ISC {2, 7, 12, 27, 52} in *isc*[9] (Step 8.2), as shown in Fig. 6f.

EXAMPLE 11. Let us consider the deallocation of the ISC $L = \{3, 9\}$ in Fig.7a. Here, $S_i = 9$, $S_i^f = 8$, *list* = {9}, and $I_i^s = \{33, 65\}$. No tree collapsing takes place here since node 8 is not free. In the process of ISC formation, node 9 combines with 5 along with the other smaller subcubes in the ISC it heads (Fig. 7b), there by ignoring our initial choice of I_i^s (Step 9.1). At the end of the deallocation phase, we have the ISCs shown in Fig. 7c.

A.2. Algorithm 2

The allocation strategy proposed in Algorithm 1 does not maintain the MSIS after every allocation, as illustrated in Example 6. This problem is eliminated in Algorithm 2. In this approach, each node maintains a list of **all** ISCs the beneath it in the form of an AVL tree, the key being the number of processors in the corresponding ISCs. The AVL tree helps to maintain the polynomial time complexities. Updating the ancestors will involve updating the AVL trees associated with them. The deallocation and allocation procedures remain basically the same as in Algorithm 1 with the following modification to the allocation procedure in Step 5.3.

Step 5.3. a) Remove the minimal ISC $\{\overline{S}_1, \overline{S}_2, \dots \overline{S}_l\}$ beneath $\overline{S}_1$ (found by searching the AVL tree associated with $\overline{S}_i$) from the corresponding *isc* list. Update all the ancestors of $\overline{S}_1$.

The proposed noncubic algorithms perform extremely well for cubic allocation too. This has been confirmed by the simulation results presented in the next section. Although the objectives of maintaining MSIS and MSS [13] may seem to be contradictory in many cases; maintaining the MSIS often acts as a "good" look-ahead allocation scheme for cubic allocation and outperforms the MSS-based strategies [12].

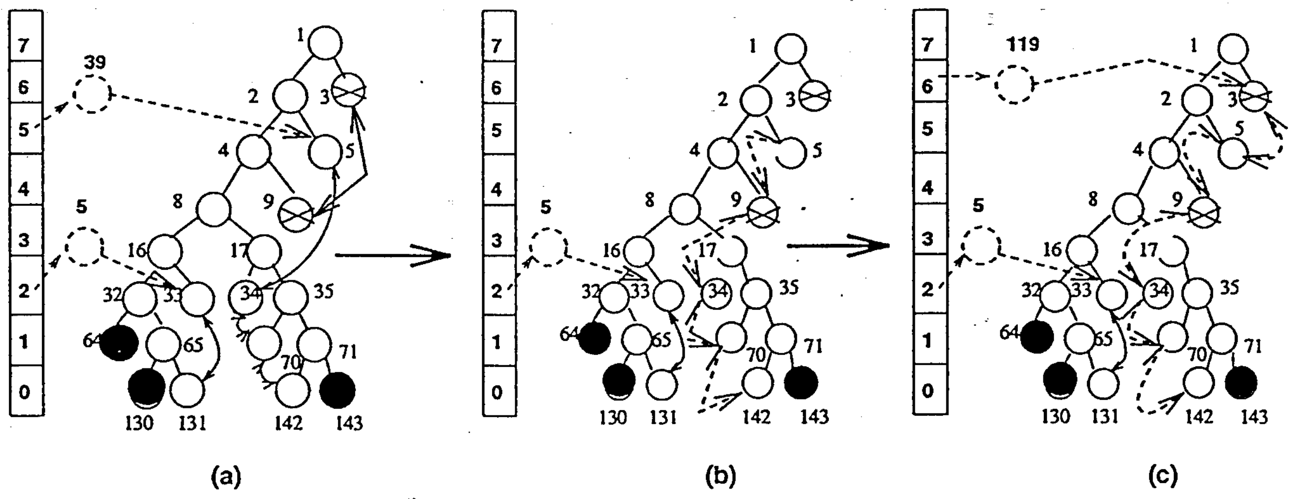


Fig. 7. Deallocation in a 10D hypercube.

TABLE I
ANALYSIS OF VARIOUS CUBIC ALLOCATION STRATEGIES

Strategy	No Cubes	Allocation	Deallocation	Memory	Type
Buddy	$2^{n+1}-1$	$O(n)$	$O(n)$	$\Theta(2^n)$	first-fit
Gray	3.2^n-3	$O(2^n)$	$\Theta(2^k)$	$\Theta(2^n)$	first-fit
M. Gray	3^n	$O\left(C^n_{\lfloor\frac{n}{2}\rfloor}\cdot 2^n\right)$	$\Theta\left(C^n_{\lfloor\frac{n}{2}\rfloor}\cdot 2^n\right)$	$\Theta\left(C^n_{\lfloor\frac{n}{2}\rfloor}\cdot 2^n\right)$	first-fit
TC	3^n	$O\left(C^n_{\lfloor\frac{n}{2}\rfloor}\cdot 2^n\right)$	$O(2^n)$	$\Theta(2^n)$	first-fit
Mod. Buddy	$n.2^n+1$	$O(n.2^n)$	$\Theta(2^k)$	$\Theta(2^n)$	first-fit
MSS	3^n	$O\left(2^{3^n}\right)$	$O(n2^n)$	$O(n3^{2n})$	best-fit
PC Graph	3^n	$O(n^{-2}.3^{3n})$	$O(n^{-2}.3^{2n})$	$O(n^{-1}.3^{2n})$	best-fit
Free List	3^n	$O(n^2)$	$O(n^2 2^{2n-2k})$	$O(n2^n)$	best-fit
Tree List	3.2^n-3	$O(n)$	$O(n)$	$O(2^n)$	best-fit
Proposed-1	$2^{n+1}-1$	$O(n^2)$	$O(n^2)$	$O(2^n)$	**best-fit**

B. Analysis of the Proposed Strategies

The allocation and deallocation time complexities of Algorithm 1 are $O(n^2)$ each and that for Algorithm 2 are $O(n^3)$ each. The space complexity of Algorithm 1 is $\Omega(n)$ and $O(2^n)$ and Algorithm 2 is $\Omega(n)$ and $O(n2^n)$. The O, Θ, Ω notations used here are the same as in [19]. The time complexities of allocation and deallocation along with the space complexity of our algorithm is compared with the existing algorithms in Table I. Depth indicates the number of adjacent subcubes that can form an ISC. The time complexity to search for an element, insert (or delete) an element in an AVL tree (be it one of *isc* lists, or the AVL list associated with each node to denote ISCs beneath it as in Algorithm 2) is $O(logN)$ where N is the number of elements in the AVL tree [19], [17]. In an *isc* list of dimension k (or the AVL tree associated with a node of dimension k) there can be $O(2^{n-k})$ elements and hence any of these operations require $O(n-k)$ or $O(n)$ time. An outline of the derivation follows; detailed derivation appears in [12].

The allocation time complexities of $O(n^2)$ and $O(n^3)$ in Algorithms 1 and 2, respectively, can be obtained as follows. Steps 1 and 2 require $O(n)$ time each. Step 3 requires $\Theta(1)$ time. Step 4 requires $O(n)$ time due to $O(n)$ possible iterations. Steps 5 and 7, each contribute $O(n^2)$ time for Algorithm 1 and $O(n^3)$ time for Algorithm 2. Step 6 requires $O(n)$ time for Algorithm 1 and $O(n^2)$ time for Algorithm 2. The deallocation time complexities can be derived as $O(n^2)$ for Algorithm 1 and $O(n^3)$ for Algorithm 2 from Step 8 which requires $O(n)$ time for Algorithm 1 and $O(n^2)$ time for Algorithm 2, with $O(n)$ possible iterations of Step 8. All the other steps contribute to lower time overheads.

The space complexity can be derived by recognizing that the dynamic binary tree may (in the worst case) have $2^{n+1}-1$ nodes. The space overhead of Algorithm 1 is $O(2^n)$ since each node has a constant overhead. Algorithm 2 requires nodes to maintain AVL tree of all free ISCs. Since there are less than 2^n free ISCs and each ISC can contribute only one entry to the AVL tree of each of its ancestors, the space overhead of Algorithm 2 is $O(n.2^n)$.

The following theorems delineate some of the properties of the proposed strategies. Proofs of these theorems appear in the appendix.

THEOREM 2. *Algorithm 1 maintains the MSIS of type SISC after every deallocation.*

THEOREM 3. *Algorithm 2 maintains the MSIS (of type SISC) after every allocation.*

THEOREM 4. *Algorithm 2 maintains the MSIS (of type SISC) after every deallocation.*

THEOREM 5. *Algorithm 2 maintains the MSIS (of type SISC) all the time.*

THEOREM 6. *The number of ISCs recognizable by both the proposed algorithms is* 2.4^n.

V. SIMULATION RESULTS AND PERFORMANCE ANALYSIS

The performance of the two proposed strategies is compared to that of free-list[2] and the modified buddy strategy. Simulation results obtained in [6] and [24] suggest that multiple gray code, tree collapsing, prime cube graph and free list exhibit similar performance for cubic allocation. Moreover, for noncubic allocation, the free-list strategy recognizes the same number of ISCs as the PC-Graph strategy if the search depth is 2 and possesses complete ISC recognition capability for depth n (Table II). Therefore, the free list strategy appears to be a typical representative of a number of allocation strategies. We use both the depths of 2 and n for simulating the performance separately. However, we present the results of depth n only as it possesses a lower waiting time than depth 2 (up to 25% less). Because it is a bit-mapping strategy, the modified buddy is also compared with the proposed strategies. Simulation results demonstrate that our schemes outperform the other two strategies.

The simulator is written in C and is run on a DEC-Station 5000. The actual CPU time required for each allocation and deallocation is measured in seconds and the system clock is advanced accordingly. The simulation is event-driven, the events being allocation and deallocation of tasks. The parameters of interest are:

1) the average waiting time of a job before it is assigned the required number of processors to execute,
2) the average time required for performing an allocation,
3) the average time required for performing a deallocation, and
4) the amount of memory required.

The following assumptions are made:

- Interarrival time between tasks: *exponential.*
- Task service time: *exponential* and *hyper-exponential.*
- Scheduling of tasks: FCFS.
- ISC (Subcube) size requested by a task: *uniform.*

Our interest is primarily in the steady-state behavior of the system under different allocation strategies. Each run of the simulation performed at least 500,000 allocations and 500,000 deallocations.

One set of simulations was performed by changing the dimension of the hypercube for the set of parameters given in Table III. It can be seen that the proposed strategies perform the best in terms of the average waiting time of a job (Fig. 8). The modified buddy strategy performs poorly even for low dimensions of the hypercube, presumably due to its first-fit nature and its poor ISC recognition capability. The free list strategy does not perform well when the search depth is 2. However, for the depth of n, the free list strategy is comparable to the proposed strategies for low dimensions. However, as the dimension of the hypercube increases, the performance of the free list with depth n starts degrading to the extent that it has a delay of about 4.5 times more than that of modified buddy and about seven times worse than the proposed strategies for a 16-dimensional hypercube. This can be attributed to the heuristics used in deallocation which do not perform well when the dimension increases and to the associated time overheads in allocation and deallocation. The variance of the modified buddy scheme is much higher than the proposed strategies (Fig. 9). The variance of FL is comparable for low dimensions, but for high dimensions it performs poorly with variance being approximately 100 times worse than ours for a 16D hypercube. This suggests a higher predictability in waiting times in addition to the lower waiting times for our strategies when compared to the other strategies.

A second set of simulations were performed assuming two-

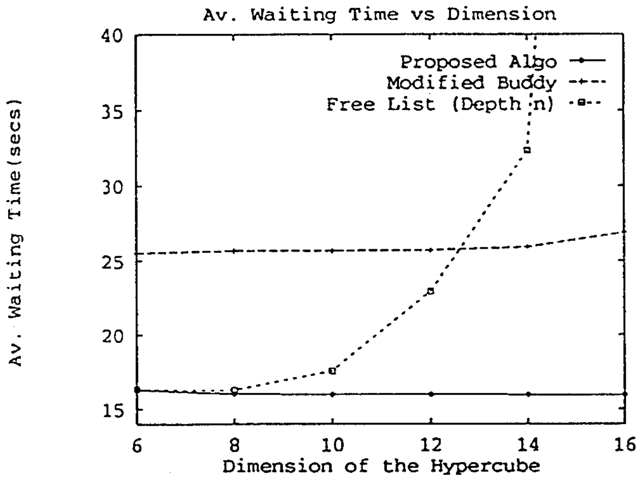


Fig. 8. Average waiting times for uniform distribution.

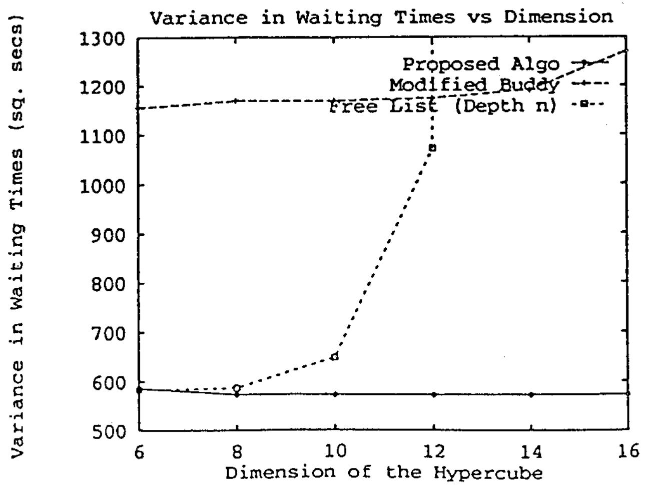


Fig. 9. Variance in waiting time vs. dimension.

2. The procedure for free list was provided by J. Kim and C.R. Das.

TABLE II
ANALYSIS OF VARIOUS NONCUBIC ALLOCATION STRATEGIES

Strategy	Depth	No ISCs	Allocation	Deallocation	Memory	Type
Buddy	n	$(n-2).2^{n-1}+3$	$O(2^n)$	$O(2^n)$	$\Theta(2^n)$	first-fit
Gray	n	$n.2^{n+1}-3.2^n+1$	$O(2^n)$	$O(2^n)$	$\Theta(2^n)$	first-fit
M. Gray	n	$3^n+2^{2n}-5.2^{n-1}-1$	$O\left(C^n_{\lfloor \frac{n}{2} \rfloor}\cdot 2^n\right)$	$O\left(C^n_{\lfloor \frac{n}{2} \rfloor}\cdot 2^n\right)$	$\Theta\left(C^n_{\lfloor \frac{n}{2} \rfloor}\cdot 2^n\right)$	first-fit
Mod. Buddy	n	$n^2.2^n-2^{n+1}+1$	$O\left(C^n_{\lfloor \frac{n}{2} \rfloor}\cdot 2^n\right)$	$O(2^n)$	$\Theta(2^n)$	first-fit
TC	n	$3^n+2^{2n}-5.2^{n-1}-1$	$O\left(C^n_{\lfloor \frac{n}{2} \rfloor}\cdot 2^n\right)$	$O(2^n)$	$\Theta(2^n)$	first-fit
PC Graph	2	$n.2^{2n-1}-n.2^n-(n-1).3^n+3n$	$O(n^{-2}.3^{3n})$	$O(n^{-2}.3^{2n})$	$O(n^{-1}.3^{2n})$	first-fit
Free List (Depth 2)	2	$n.2^{2n-1}-n.2^n-(n-1).3^n+3n$	$O(n^3.3^{2n})$	$O(n^3.3^{2n})$	$O(n.2^n)$	first-fit
Free List (Depth n)	n	$O((n!)^2)$	$O\left(n^{-n}.2^{n^2}+n^3.2^{2n}\right)$	$O(n^3.3^{2n})$	$O(n.2^n)$	first-fit
Proposed-1	n	2.4^n	$O(n^3)$	$O(n^2)$	$O(2^n)$	best-fit
Proposed-2	n	2.4^n	$O(2^3)$	$O(n^3)$	$O(n.2^n)$	optimal

TABLE III
PARAMETERS USED IN SOME SIMLUATIONS

Interarrival time of tasks	Service Time of Tasks	ISC Size Distribution
Exponential	Exponential	Uniform
Mean = 11 secs	Mean = 10 secs	—

phase hyper-exponential distribution [22] of job residence times. The two phase hyperexponential distribution consists of two exponential distributions where the residence time is generated from the first exponential distribution (mean μ_1) with a probability p and from the second distribution (mean μ_2) with a probability $1-p$. For our simulations, we have assumed $\mu_1 = 10$ *secs*, $\mu_2 = 90$ *secs*, $p = 0.9$. Fig. 12 presents the relative average waiting delay as a function of dimension for an utilization of 0.4. The utilization of the system can be derived as $u = \frac{\bar{n}.\mu}{\lambda 2^n}$, where $\bar{n}$ represents the average number of processors demanded by a task, μ represents the average job residence time, λ represents the job arrival rate (inverse of inter-arrival time). The proposed strategy outperforms the other strategies due to our good ISC recognition capability combined with the best-fit nature of our scheme, as described earlier.

A third set of simulations were performed by changing the rates of arrival for a 12D hypercube while keeping the other parameters the same as the first set of simulations. Results for various dimensions indicate that our strategy outperforms the other two under all traffic conditions due to the reasons mentioned above. Fig. 10 shows the response of average delay for various arrival rates for a 12D hypercube. The trends were similar for hypercubes of other dimensions and for the pseudo-normal distribution of task sizes as well. (The pseudo-normal distribution a triangle distribution, mimicing a normal distribution [8], [12].)

A fourth set of simulations were run assuming a job-mix of both cubic and noncubic requests. The probability of having a cubic request was varied for a 12-dimensional hypercube (since the FL exhibits poor performance for higher dimensions). The parameters for the arrival and job residence times are the same as in Table III. The average waiting delays are shown in Fig. 11. It can be observed that our scheme outperforms the FL strategy. However, the difference in performance diminishes as the number of (exclusively) cubic requests increases. This confirms the earlier observations [6], [7], [20] that all the allocation strategies behave the same for cubic allocation. However, our simulations indicate that when noncubic requests are involved, an efficient allocation strategy may significantly improve the performance in hypercubes.

In all the four sets of simulations our strategy had the least execution times for both allocation and deallocation. Fig. 13 presents results for the first simulation set. The free list performs the worst, which is understandable, given the time complexities in Table II. (It should be noted that the goal of the free list strategy was to provide a strategy with complete recognition and best-fit search capability for cubic allocation, at the expense of high time and space overheads.) The modified buddy also has a higher average allocation and deallocation time than the proposed strategies, as expected. The free list had a large average deallocation time (e.g., more than 2,000 times than ours for a 16-dimensional hypercube). The allocation and deallocation times for both the proposed strategies are almost identical.

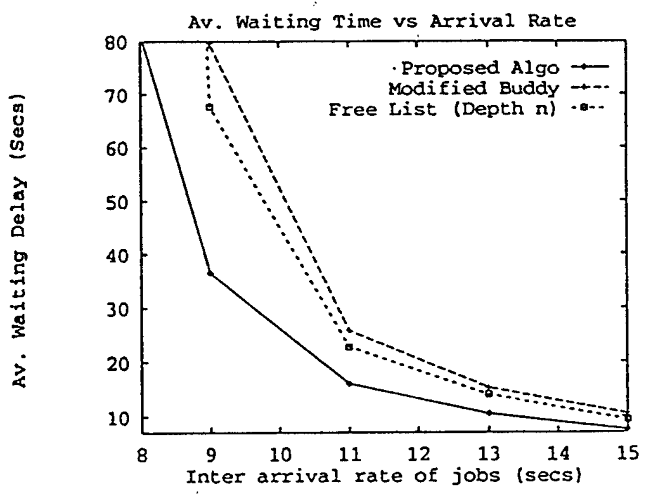


Fig. 10. Uniform distribution for a 12D hypercube.

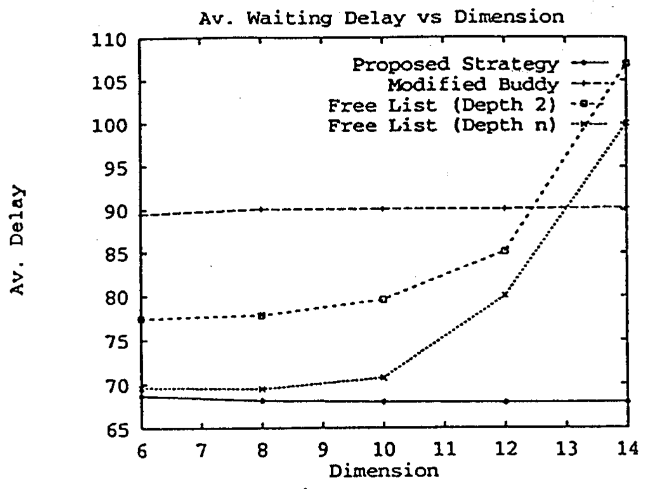


Fig. 12. Average waiting delay vs. dimension.

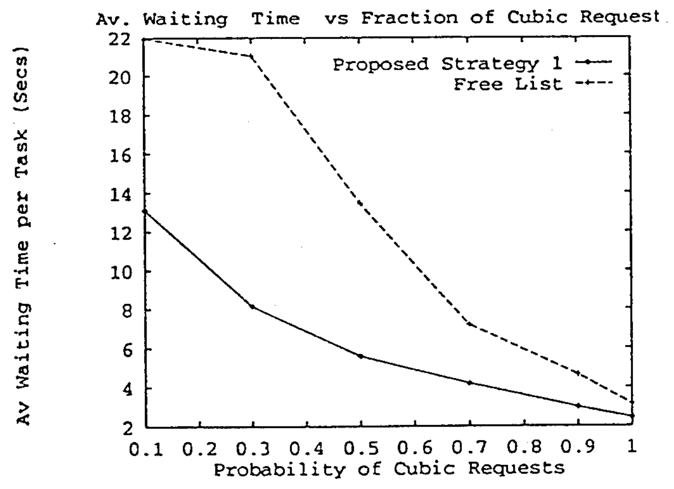


Fig. 11. Average waiting times for various job-mixes.

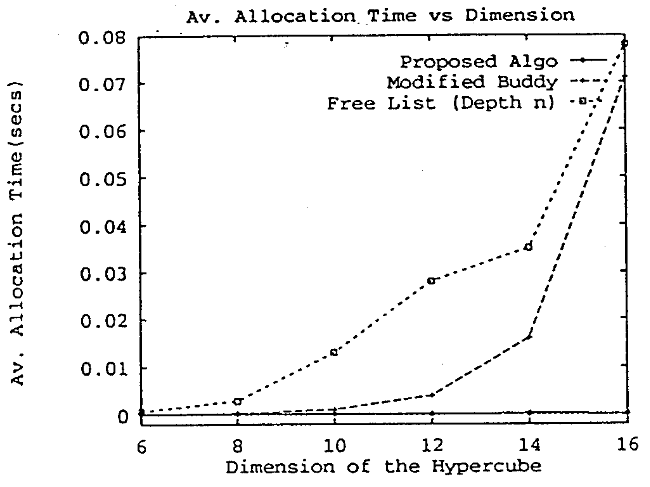


Fig. 13. Average allocation time vs. dimension.

In all the four sets of simulations both our strategies performed identical with respect to delays and the (de)allocation times. The difference in time complexities arises because the second algorithm keeps track of all the ISCs beneath a node whereas the first algorithm just keeps the maximal ISC. Possibly, for the sizes of hypercubes we have considered, not many nodes in the dynamic binary tree have many ISCs beneath them to make an impact in the performance and allocation/deallocation time. It may be possible that if we further increase the dimension of the hypercube we may see some difference in their performance under high arrival rates.

The maximum amount of memory required by the various strategies were also measured for the various simulations performed. (The memory requirements was measured by keeping track of the number of "malloc's and "free's performed and adding that to the fixed memory used by the strategy.) Figs. 14 and 15 represent the memory requirements for the various strategies assuming the same set of parameters given in Table III. For low dimensions (Fig. 14), the modified buddy has the lowest memory requirement as the overhead of maintaining the free lists in terms of pointers and other information per node in the dynamic binary tree dominates. However, as the dimension of the hypercube increases, our strategy has the least memory requirement, as the dynamic binary tree saves us the space by pruning the nodes that are not needed. The Free-list has the worst memory requirement as it needs to form all possible subcubes from a released subcube. This scenario does not change even under extremely high traffic rates (Fig. 15).

From the simulations it can be said that our strategy performs the best in terms of all the four parameters of interest for both cubic and noncubic requests. Although all the strategies result in the same average waiting delay for (only) cubic requests, as reported in earlier studies [20], [23], the average waiting delay improves significantly in the presence of noncubic requests. For noncubic allocation, an efficient processor allocation scheme may reduce the waiting delays significantly, unlike cubic allocation. This is similar to the submesh allocation problem in meshes [9]. It can be attributed to the fact that in noncubic allocation, there can be requests for 2^n sizes and the nature of the allocation policy may have a great impact on subsequent requests; unlike the cubic allocation where the nature of the allocation policy does not have a considerable impact on subsequent requests because of the limited number ($n + 1$) of requests possible. Though our ISC recognition capability could be better, the low time complexities and our capability to effectively maintain the MSIS of ISCs of type SISC after every allocation and deallocation along with our lower memory requirement for high dimensions makes it extremely effective. This effect becomes more prominent as the dimension of the hypercube increases. Further performance improvements may be possible by using an efficient job scheduler along with the proposed processor allocator, as demonstrated for cubic requests in hypercubes [20] and in meshes [10] or by using a time-sharing strategy [11].

VI. Conclusions

Sizes of manufactured hypercubes are increasing, expected to continue so as demand for parallel computation increases. Currently, nCUBE manufactures 8,192 node hypercube systems and is targeting for 65,536 node hypercube systems for tera-flops performance [21]. As these sizes grow, it will become virtually impossible to use near optimal algorithms with exponential and super-exponential time complexities, given limitations on processor speed imposed by the technology.

Implementing relatively efficient allocation and deallocation algorithms with very low time and space complexities becomes desirable so host computers do not perform only allocations and deallocations. This will ensure lower turn-around times for individual tasks, as well. The proposed scheme is, indeed, a practical solution with its polynomial time overheads for both allocation and deallocation, its a low memory overhead and its lowest average waiting times for jobs among all existing strategies.

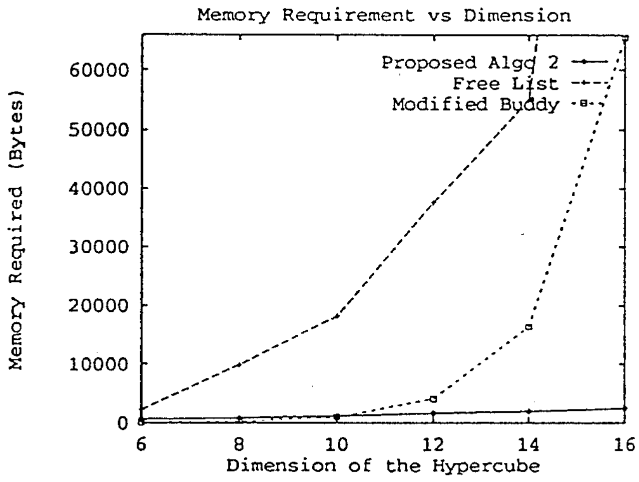


Fig. 14. Memory requirement of various strategies.

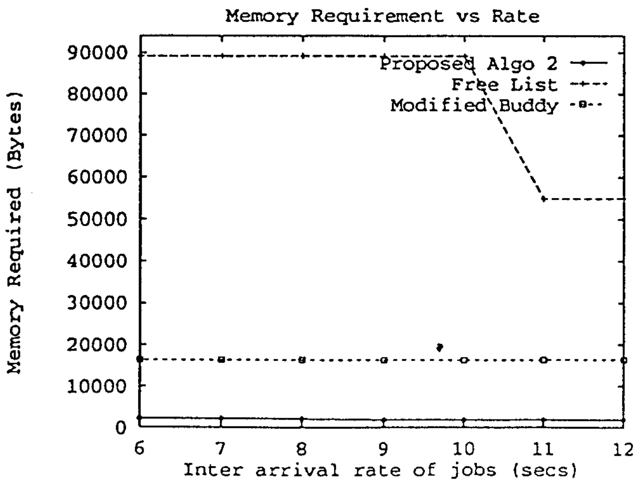


Fig. 15. Memory requirement of various strategies vs. job arrival rate for a 14D hypercube.

Appendix

Theorem 1. *SISC is an ISC.*

Proof. From definition 5, if $S = \{S_1, S_2, ..., S_m\}$ is a SISC, the sibling of S_i is the common ancestor of $\{S_{i+1}, ..., S_m\}$. Hence all the subcubes in $\{S_{i+1}, S_{i+2}, ..., S_m\}$ are at Hamming distance 1 from S_1 with exact distance between S_i and S_j as $d_i - d_j + 1$ ($j > i$). This is because, since all the subcubes are represented in the binary tree their xs appear to the extreme right and the Hamming distance between any node and its sibling (or the sibling's descendants) is 1. Thus any two subcubes in S will meet the Hamming distance and exact distance criteria mentioned in Definition 3.

Theorem 2. *Algorithm 1 maintains the MSIS of type SISC after every deallocation.*

Proof. Suppose during deallocation there is a subcube headed by S' that could not be combined with the higher dimensional subcube S (and is a descendant of the sibling $\overline{S}$ of S), but is of higher size than what S is combined with to form an ISC. This is not possible given that while combining S with lower dimensional subcubes we check for the highest ISC beneath $\overline{S}$ to combine with S. Hence if S' is the highest incomplete subcube beneath $\overline{S}$ it would have combined with S. Thus the algorithm maintains the MSIS (of type SISC) after every deallocation.

Theorem 3. *Algorithm 2 maintains the MSIS (of type SISC) after every allocation.*

Proof. Suppose the Algorithm 2 chooses an ISC $S = \{S_1, S_2, ..., S_i, S_{i+1}, ..., S_m\}$ whereas an ISC $S' = \{S_1, S_2, ..., S_i, S'_{i+1}, ..., S'_p\}$ with fewer nodes could satisfy the request. (S and S' match in the first i subcubes ($i \geq 0$)). Say $i = 0$. So Algorithm 2 has chosen an ISC from the *isc* list that is not the least sized ISC to satisfy the request. This contradicts Step 2 of the algorithm. Thus it is not possible. Say $i > 0$, i.e., the algorithm chose the same ISC as what the optimal would have but after allocating i subcubes the proposed algorithm failed to recognize the presence of $\{S'_{i+1}, S'_{i+2}, ..., S'_p\}$ as the minimal ISC beneath the sibling of S_i that could satisfy the remaining number of nodes required. This is also impossible as we maintain all ISCs beneath a node in the form of an AVL tree and search for the minimal ISC beneath the sibling of every allocated node (Step 5.3.a). Hence such a scenario is impossible to arise. Thus the strategy maintains the MSIS after every allocation.

THEOREM 4. *Algorithm 2 maintains the MSIS (of type SISC) after every deallocation.*

PROOF. The proof of this will be exactly the same as Theorem 2 as the algorithm is also capable of recognizing the maximal ISC beneath it in every step of deallocation like Algorithm 1.

THEOREM 5. *Algorithm 2 maintains the MSIS (of type SISC) all the time.*

PROOF. Since we start with the entire hypercube (which is an MSIS) and maintain the MSIS after every allocation and deallocation (previous two theorems), the MSIS is always maintained.

THEOREM 6. *The number of ISCs recognizable by both the proposed algorithms is* 2.4^n.

PROOF. Since the proposed algorithms recognize all SISCs we have to find the number of SISCs in an n-dimensional hypercube. Say $T(k)$ represents the number of ISCs with a particular k-dimensional subcube as its head. A k-dimensional subcube can combine with all the ISCs beneath its sibling (including no ISC—in which case we have just a subcube). Thus, $T(k) = 2.T(k-1) + 2^2.T(k-2) + \ldots + 2^i.T(k-i) + \ldots + 2^k + 1$ as there can be two $k-1$ dimensional subcubes with whom the k-dimensional subcube can combine, 4, $k-2$ dimensional subcubes with whom the k-dimensional subcube can combine and so on. The 1 represents the case when no other subcube combine with the k-dimensional subcube. Expanding $T(k)$ we can see that $T(k) = 4.T(k-1)$ with the boundary condition that $T(0) = 2$ (1 for including the node and 1 for not including). The number of ISCs in an n dimensional hypercube is $T(n)$ where the last term 1 would indicate a full hypercube instead of the NULL set. Thus, $T(n) = 2.4^n$.

ACKNOWLEDGMENT

This research was supported by the U.S. Air Force Office of Scientific Research and a Texas ATP grant.

REFERENCES

[1] A. Al-Dhelaan and B. Bose, "A new strategy for processor allocation in an N-cube multiprocessor," *Proc. Int'l Phoenix Conf. Computing Comm.*, pp. 114-118, Mar. 1989.

[2] S. Baase, *Computer Algorithms: Introduction to Design and Analysis*, Addison Wesley, Nov. 1988.

[3] M.S. Chen and K.G. Shin, "Processor allocation in an n-cube multiprocessor using gray codes," *IEEE Trans. Computers*, vol. 36, no. 12, pp. 1,396-1,407, Dec. 1987.

[4] M.S. Chen and K.G. Shin, "Task migration in hypercube multiprocessors," *Proc. 16th Ann. Int'l Symp. Computer Architecture*, pp. 105-111, May 1989.

[5] M.S. Chen and K.G. Shin, "Subcube allocation and task migration in hypercube multiprocessors," *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1,146-1,155, Sept. 1990.

[6] P.J. Chuang and N.F. Tzeng, "Dynamic processor allocation in hypercube computers," *Proc. 17th Ann. Int'l Symp. Comp. Architecture*, May 1990.

[7] D. Das Sharma and D.K. Pradhan, "A novel approach for subcube allocation in hypercube multiprocessors," *Proc. Fourth IEEE Symp. Parallel and Distributed Systems*, pp. 336-345, Dec. 1992.

[8] D. Das Sharma and D.K. Pradhan, "Fast and efficient strategies for cubic and noncubic allocation in hypercube multiprocessors," *Int'l Conf. Parallel Processing*, vol. I, pp. 118-127, Aug. 1993.

[9] D. Das Sharma and D.K. Pradhan, "A fast and efficient strategy for submesh allocation in mesh-connected parallel computers," *IEEE Symp. Parallel and Distributed Processing*, pp. 682-689, Dec. 1993.

[10] D. Das Sharma and D.K. Pradhan, "Job scheduling in mesh multicomputers," *1994 Int'l Conf. Parallel Processing*.

[11] D. Das Sharma, G.D. Holland, and D.K. Pradhan, "Subcube level timesharing in hypercube multicomputers," *1994 Int'l Conf. Parallel Processing*.

[12] D. Das Sharma and D.K. Pradhan, "Novel strategies for cubic and noncubic allocation in hypercube multiprocessors," Technical Report TR-93-023, Dept. of Computer Science, Texas A&M Univ.

[13] S. Dutt and J.P. Hayes, "On allocating subcubes in a Hhpercube multiprocessor," *Proc. Third Conf. Hypercube Computers and Applications*, pp. 801-810, Jan. 1988.

[14] C. Hu, M. Bayoumi, B. Kearfott, and Q. Yng, "A parallelized algorithm for the the preconditioned interval newton method," *Proc. Fifth SIAM Conf. Parallel Processing for Scientific Computing*, Mar. 1991.

[15] J. Kim, C.R. Das, and W. Lin, "A top-down processor allocation scheme for hypercube computers," *IEEE Trans. Parallel and Distributed Systems*, vol. 2, no. 1, pp. 20-30, Jan. 1991.

[16] J. Kim, C.R. Das, and W. Lin, "A processor allocation scheme for hypercube computers," *Proc. 1989 Int'l Conf. Parallel Processing*, vol. II, pp. 231-238, Aug. 1989.

[17] E. Horowitz and S. Sahni, "Data structures in Pascal," *Galgotia Booksource*, 1984.

[18] K.C. Knowlton, "A fast storage allocator," *Comm. ACM*, vol. 8, no. 10, pp. 623-625, Oct. 1965.

[19] D. Knuth, *The Art of Computer Programming: Sorting and Searching*. Reading, Mass.: Addision-Wesley, 1973.

[20] P. Krueger, T.-H. Lai, and V.A. Radiya, "Processor allocation vs. job scheduling on hypercube computers," *Proc. 11th Int'l Conf. Distributed Computing Systems*, pp. 394-401, May 1991.

[21] *nCUBE 2 Systems: Technical Overview*, nCUBE Corp., Foster City, Calif., 1992.

[22] K.S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Englewood Cliffs, N.J.: Prentice Hall, 1982.

[23] N.-F. Tzeng, H.L. Chen, and P.J. Chuang, "Embeddings in incomplete hypercube," *Proc. 1990 Int'l Conf. Parallel Processing*, Aug. 1990.

[24] H. Wang and Q. Yang, "Prime cube graph approach for processor allocation in hypercube multiprocessors," *Proc. 1991 Int'l Conf. Parallel Processing*, vol. 1, pp. 25-32.

Debendra Das Sharma received the BTech degree in computer science and engineering with honors from the Indian Institute of Technology, Kharagpur, in 1989, and the PhD degree in electrical and computer engineering from the University of Massachusetts, Amherst, in 1995. Dr. Das Sharma was a research associate with the Department of Computer Science at Texas A&M University, College Station, from 1992 to 1994. He has been with the Research and Development Laboratory of Hewlett-Packard, Roseville, Calif., since 1994. His research interests include parallel processing, formal verification and specification of parallel architectures, computer architecture, and fault-tolerance.

Dhiraj K. Pradhan holds the COE Endowed Chair in Computer Science at Texas A&M University, College Station. Prior to joining Texas A&M, he served until 1992 as professor and coordinator of computer engineering at the University of Massachusetts, Amherst. Funded by the U.S. National Science Foundation, the U.S. Department of Defense, and various corporations, he has been actively involved—and has presented numerous papers—in VLSI testing, fault-tolerant computing, and parallel processing research—with extensive publications in journals over the past 20 years.

Dr. Pradhan served as guest editor of special issues on fault-tolerant computing for *IEEE Transactions on Computers* and *Computer* magazine, published in April 1986 and March 1980, respectively. Currently, he is an editor of several journals, including *IEEE Transactions on Computers* and *JETTA*. He served as general chair of the 22nd Fault-Tolerant Computing Symposium and program chair of the IEEE VLSI Test Symposium. Dr. Pradhan is a coauthor and editor of the book *Fault-Tolerant Computing: Theory and Techniques*, volumes I and II (Prentice Hall, 1986; second edition, 1993). He is a fellow of the IEEE and the recipient of the Humboldt Distinguished Senior Scientist Award.

V. Software Fault-Tolerance

Cooperating Diverse Experts: A Methodology to Develop Quality Software for Critical Decision Support Systems[1]

Dhiraj Pradhan
Herbert Hecht[†]
Myron Hecht[†]
Fred Meyer
Nitin Vaidya

Department of Computer Science
Texas A&M University
College Station TX 77843-3112

[†]SoHaR Incorporated
8421 Wilshire Boulevard, Suite 201
Beverly Hills CA 90211-3204

Abstract – The problem of developing software for critical systems in the decision support context is considered. The limitations of existing software development methodologies are mentioned and a new methodology, cooperating diverse experts (CDE), is proposed. This new methodology draws upon techniques used in multiple version software and in distributed recovery blocks. The methodology relies upon the ultrareliable development of a parameterizable arbitrator to administer the cooperation of multiple diverse implementations (interpretations) of the decision support problem. CDE may be used to develop a single reliable software module or it may be used as an operational system in which some modules are multiply implemented.

I. Introduction

The lack of reliable software for critical systems has haunted aerospace for decades: the launch failure of the Mariner I in 1962, fly-by-wire aircraft, the NASA space shuttle, and the NASA Magellan spacecraft. A variety of approaches have been espoused over the last two decades to address the problem [25]. Much study has been directed toward the ultimate goal of automatic code generation [5, 21]. Figure 1 illustrates the principle.

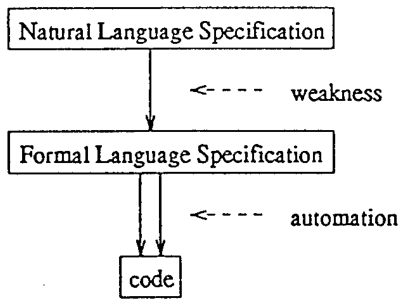


Figure 1: Automatic code generation

[1]Research reported supported in part by AFOSR F49620-94-1-0276 and RADC F30602-94-C-013.

Any programming project must commence from some natural language specification (or understanding) of the problem. This methodology entails adopting, essentially, a very high level (formal) language. The automatic code generation ensures that if the specification in the formal language is correct then the resultant code is correct. The difficulty is that the use of the formal language involves one or more of the following problems:

1. The code generation is not entirely automatic, invalidating the assurance of correct code.
2. The formal specification language covers only a narrow aspect of application domains (so most of the specification will be written in an error-prone high level language).
3. The formal specification language is considerably abstract and complex; this leads to human error in translating from the natural language specification to the formal specification.
4. The formal language does not capture certain requirements, for example the timeliness of the outputs. (For our purposes, let us define the requirements to be anything not captured by the formal specification.) Once the generated code is modified to satisfy the requirements, the guarantees of the automatic code generation are lost.

To address these problems, an opposite approach may be used. This allows the code to be freely modified. The code is validated by an automated tool that shows whether the code matches an independently developed formal specification. Figure 2 diagrams the method. Essentially, the tools derive condition tables from the code, which are then compared to the condition tables of the specification.

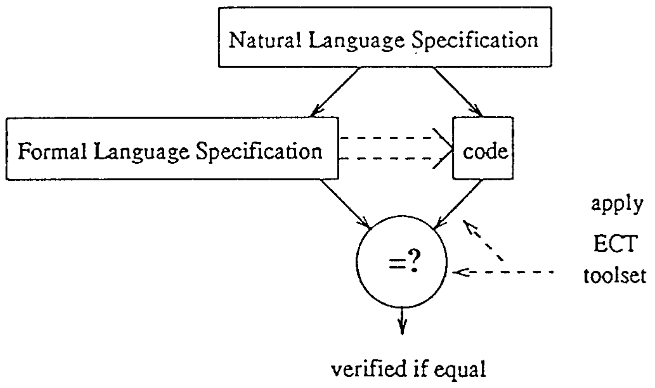


Figure 2: Enhanced Condition Tables (ECT)

Two representative examples of the approach are described in [1, 7]. Both use condition tables [6] as the formal specification language. More labor in development is needed than with automatic code generation. The assurance of correctness for this approach is directly related to the degree of independence of code development from condition table development. To the extent that there is an implication in the development process, from the formal language specification to the code, the testament to the code's validity is weakened.

SoHaR has developed a toolset that consists of tools to semi-automatically parse C or Ada code, form condition tables, and formulate rules to resolve the don't cares (more generally, any flexibility) in the test set (embodied by the formal specification) to exercise special values. The toolset is currently being used to develop reusable fault-tolerant components in Ada. For example, software to administer distributed recovery blocks [11, 20] has been validated using the ECT toolset.

While these and other methodologies show promise, they have additional limitations when trying to cope with programming projects for expert systems or for decision support systems. In the next section we discuss the nature of decision support systems and the appropriateness of adopting a multiple version software (MVS) [4] approach. In Section III, we describe the CDE software development process and give a preliminary discussion of the issues involved. A testbed is under development at Texas A&M University. Section IV covers some initial concerns about the CDE methodology that the testbed will address. We summarize these concerns in Section V.

II. Software with Diverse Interpretations

This work pertains to a development process to arrive at reliable software for critical knowledge based systems. These software systems have inherent problems due to inconsistencies that may exist in their design or in the knowledge base. Since it is unduly complex to attempt to resolve these inconsistencies before proceeding with the software development, we have devised an approach that allows for the software development to proceed despite inconsistencies. The process of testing the resultant software then aids in the identification and resolution of specification or knowledge base inconsistencies.

Expert systems are appropriate for aerospace applications involving so-called 'soft' problems under time constraints such that there can be no significant review by human experts. These soft problems can include those with demanding real-time constraints, such as threat assessment, selection of countermeasures during an engagement, and radar track estimation and prediction. Also, when real-time constraints are not demanding, then decision support systems may still be appropriate if there is a significant volume of information to be processed. For example, even with no strict real-time constraints, there may not be sufficient time for human experts to draw detailed conclusions about a significant number of what-if scenarios.

Methodologies and analytical procedures that enable one to rely on a software product are of interest due to the cost-intensive and life-critical nature of aerospace systems; but no adequate methodologies and procedures have been developed. In particular, verification and validation techniques for expert systems remain more art than science. While the inference engine (reasoning methods) of a software product can be partially validated, there is no effective methodology to establish the integrity of the knowledge base or to validate the connections between the knowledge base and the inference engine.

Methods to overcome these difficulties have been proposed [18, 22], but have not been demonstrated effective. The results of the research efforts for NRC and EPRI, as assessed in the SAIC report [16], are not conclusive at all. Therefore, we believe that implementation of multiple expert systems and subsequent voting or reasoning is the best available technique to achieve high confidence in the dependability of the software product for these applications. MVS is applicable to expert systems, unlike most formal methods, and has achieved notable results [9]. MVS is sufficiently accepted that the FAA reduces the testing requirements of individual versions when they are developed as a MVS system.

A prototypical architecture for systems with multiple instantiations of software modules is shown in Figure 3. There is a broadcast mechanism to ensure that each instantiation of each module functions on the same inputs and a decider mechanism to determine the proper module outputs. The problem of reaching a consensus on input values (such as sensor readings) is well studied [12, 14, 24] and will not be considered here. Architectures such as FTMP [8], SIFT [26], and MAFT are meant to treat hardware and communication errors—they generally presuppose that any multiple instantiations of an invocation of a module are actually replicates of a single software implementation of that module. We will consider software architectures where multiple versions of each module are written [4].

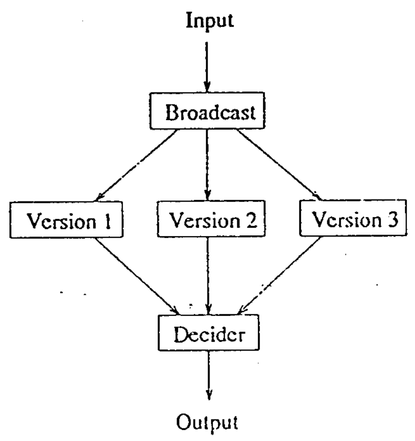


Figure 3: Multiple version architecture

When multiple versions of a module are written, the diversity of the versions is the principal goal. Research investigations have concentrated on achieving program diversity by diversifying various elements of the software development process. For example, experiments have used different processors, different programming languages, and different data structures (e.g., fixed point in some versions and floating point in others).

These design diversity experiments revealed that for well implemented multiple version software (MVS) systems, the bulk of faults not tolerated by the system can be attributed to those portions of the software development, such as creating the formal specification, that were not multiply implemented. Diversifying the specifications is meant to accommodate these problems. So one aspect of diversity sought is at the specification level [2, 10]. This can include using diverse specification languages in the hope that different types of errors in translating natural language specifications to formal specifications or in translating formal specifications to code will not be correlated among the versions.

At this point, consider that there are two ways to view MVS systems. One view is with the objective of creating an operational system with the multiple instantiations mapped onto one or more processors. The other view is to consider each diverse phase of the software development effort as an opportunity to identify errors introduced in that phase or extant from previous phases. For example, in going from formal specifications to code, comparison of the test results for the different versions can identify coding errors and can also lead to the discovery of errors or inconsistencies in the specification [3]. Similarly, if the formal specifications are diverse, then comparison of the versions could detect errors made in arriving at the formal specifications and could also lead to the discovery of unclear or inconsistent natural language specifications.

If one imagines a continuum from requirements to code, the MVS philosophy encourages pushing diversity so it is as close to the requirements as possible. There is a limitation on this diversity due to the necessity in Figure 3 of defining a means to compare the outputs of the diverse implementations. Our proposed software architecture achieves comparison of outputs by having the implementations provide facilities to submit proposed actions (outputs) to each other and evaluate the proposed actions of others. Figure 4 shows the path of the CDE methodology from the natural language specification to an operational system.

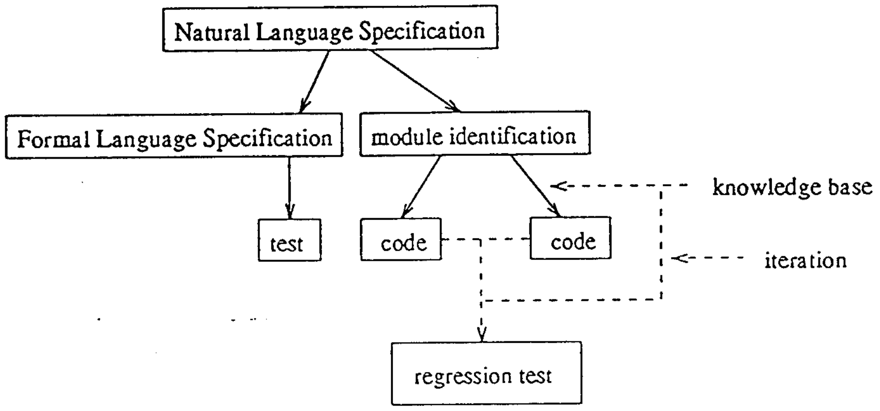


Figure 4: Cooperating diverse experts

CDE does not address the testing problem. It is assumed that an initial test set is obtained from some formal language specification. Initially, the problem is broken down into fairly large modules and each module may be implemented by multiple independent programming teams. The bulk of the (potentially inconsistent) knowledge base does not come into play until multiple interpretations are being coded.

Back-to-back testing is conducted on the multiple interpretations and, for each failure of an acceptance test noted, a determination is made placing the discrepancy into one of four categories:

1. The test result expected was not valid. The formal specification requires modification.
2. There is a coding error in one of the interpretations—rectify.
3. There is an inconsistency in the knowledge base with a clear resolution. Correct the knowledge base.
4. There is an inconsistency in the knowledge base without a clear resolution. Make the formal specification more detailed to generate the additional tests needed to assess the circumstances; adjust code as appropriate.

In addition, the back-to-back testing results can be used to accumulate a set of regression tests. Some of the discrepancies (especially case 4 above) are indicative of tests that one should ensure are retained when the specifications are modified during software maintenance.

By relaxing the voting restriction, an additional layer of diversity is possible. The multiple implementations may now be functionally diverse. This action-event model is very flexible in that interpretations may take entirely different approaches to solving the problem. It also, regrettably, puts considerable demands on the interpretations, for they must be able to accept events (outputs from other modules) and to assess actions proposed by other interpretations. These actions may take on fairly arbitrary forms. Also, it is no longer necessary for the software requirements (or natural language specification) to be complete, correct, and consistent at the outset.

As a result, the benefits of the cooperating diverse experts approach are most succinct for 'soft' problems. When a particular problem is not well understood or has associated with it a large and presumably inconsistent human knowledge base, then it is hard to identify the inconsistencies and it is even harder to confidently take measures to resolve them. In CDE systems, since the constraints of the *decider* are somewhat pushed away, it is easier for the human knowledge inconsistencies to propagate to the code where they can be identified. The inconsistencies discovered can then be resolved at the same time that there is greater understanding of their effects, because code has already been written and is being tested. (Or some inconsistencies can be left in if it is believed that they reflect the natural differences in human expert opinion on the problem.)

III. CDE Development Process

An outline of the phases in a software development effort is shown in Figure 5. A description of each phase and its relation to the CDE methodology follows:

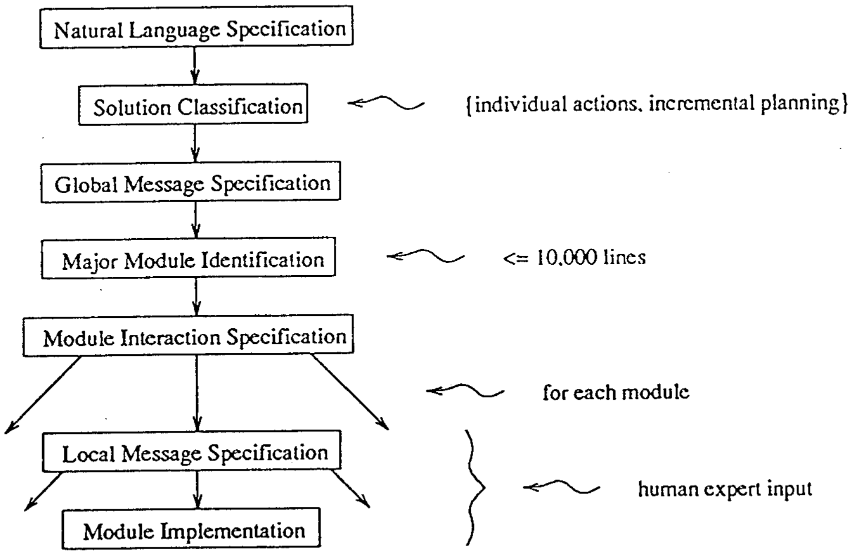

Natural Language Specification
Solution Classification
{individual actions, incremental planning}
Global Message Specification
Major Module Identification
<= 10,000 lines
Module Interaction Specification
for each module
Local Message Specification
Module Implementation
human expert input

Figure 5: Specification hierarchy

Natural Language Specification. This is the (English) language formulation of the problem and the system requirements. It might not be necessary (and perhaps not appropriate) for this to be a detailed specification.

Solution Classification. An obvious dichotomy occurs when one considers whether the problem is naturally solved by means of specifying a plan and then continually improving the plan (or adapting it to changing inputs). We refer to this classification as *incremental* *planning.* The alternative is to slice time into small intervals and to consider changing any (or all) possible outputs for each time interval (or continuously) without long-range planning. Since we refer to all module outputs as actions, we refer to this classification as *individual actions.* This is the classification generally assumed for hard real-time systems. Other classifications may also be appropriate.

These classifications are mentioned because they have an effect on the global message specification required to provide appropriate facilities to the cooperating diverse experts. For complex software projects, different classifications may seem appropriate for the various modules identified. The sole purpose of solution classification is to determine the aggregate facilities that must be met by the global message specification.

Global Message Specification. The purpose of this stage is to specify the tools necessary to implement the cooperating independently written versions of each module. The solution classification dictates to some extent the variety of process control and recovery facilities that will be needed. The development process is still largely dominated by data structure considerations. The physical objects (nouns) that the problem models and their possible qualities (adjectives) may be fully characterized at this level.

Major Module Identification. Assuming that the problem is complicated, it is necessary to divide it into modules; preferably, with little anticipated interaction between modules. Each module will be implemented by more than one programming team with minimal interaction between teams allowed. Since the specifications are necessarily vague and the problem is hard, the programming teams can be expected to arrive at different solutions. We call these solutions interpretations. This is the first point in the development process where verbs (e.g., methods) come clearly into play. It is perhaps disadvantageous to break down the problem into modules whose complexity is much less than an estimated 10,000 lines of code in a procedural language. Every stage of dissecting the problem leaves less to be coded by multiple interpretations (and more that is globally specified—single point of failure).

Module Interaction Specification. It is important that, at this point in the development, as much flexibility as possible remains available to the development teams that will implement the diverse interpretations. This maximizes the benefits to be gleaned from diversity. For example, Figure 6 shows an edge-weighted graph. Suppose that part or all of the module's purpose is to compute a minimal-cost path from the position, P, to a destination, D. The minimal-cost path for this trivial example is shown in bold. The cost of each edge traversal may reflect time or some other undesirable 'cost,' such as hazard (for an escape problem).

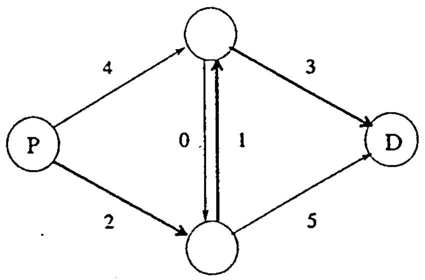


Figure 6: Finding a minimal-cost path

The software project benefits with respect to diversity if the programming teams are not restricted in the algorithms they choose. One team may choose a best-bud first (breadthfirst search) approach; another team may opt for a branch-and-bound (depth-first search) approach; and another team may solve the problem by computing the transitive closure of the adjacency matrix of the graph.

Local Message Specification. Most of the human knowledge base concerning the problem is considered here and during module implementation. At this level, any additional facilities needed by the independent software versions are specified. These facilities can provide for the exchange of intermediate results (conclusions).

Module Implementation. The several interpretations of each module are all run on the computing resources available. The various interpretations of the various modules may be mapped to 100 processors or to 1. Some design decisions may be affected by this, but the effects on the technical correctness of the CDE methodology are presumed small and will be ignored in this discussion. As long as facilities are provided for preemptive interrupts and the detection of protection faults, catastrophic results will not ensue due to shared processors. An interpretation may even implement a section of its 'code' as multiple interpretations—spawning the appropriate processes as needed.

Figure 7 shows one choice for the allowed interactions between the interpretations of a module. Each interpretation, during the course of its computations, arrives at *actions* that it believes to be advantageous. Here, action may mean a scalar action (individual actions) or a change to the current plan (incremental planning). Since software errors may occur, the interpretation submits its proposed action to the arbitrator, which chooses another interpretation to examine the proposed action. If the latter interpretation approves the action, then the arbitrator is informed.

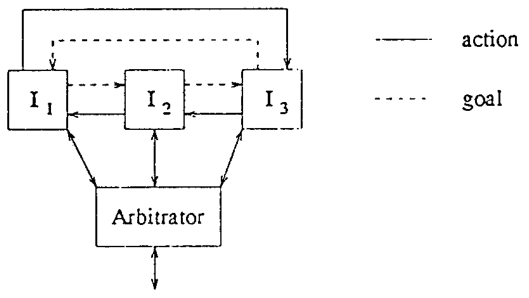


Figure 7: Interaction of diverse experts

The arbitrator is a comparatively simple piece of code that may be implemented with confidence. Furthermore, the implementation of the arbitrator is essentially independent of the local message specification—that is to say, it consists largely of reusable code. If the number (see below) of the approved action matches the number expected and if the action descriptions sent to the arbitrator from the two agreeing interpretations match, then the arbitrator declares the action taken and duly notifies all the interpretations as well as any appropriate external processes. Numbering actions is appropriate, because another approved action or external event may have taken place in the intervening time and the approved action is not assured to be appropriate under these circumstances.

The static selection of açceptance testers depicted by the loop graph in Figure 7 is suitable for tolerating one software 'error' at a given instance. For static testing graphs, determining the degree of error tolerance is a generalization of the thoroughly studied system-level diagnosis problem [13, 19, 23]. The arbitrator may also implement various flexible procedures to choose acceptance testers. For example, an idling process may be preferred as an acceptance tester. Also, the multiple interpretations need not be run simultaneously. The arbitrator may adopt an algorithm to spawn interpretations as needed. See [17] for a discussion of these issues.

Since the interpretations are purposefully diverse, it is natural to be concerned that submitted and approved actions might be unduly rare. To remedy this, a facility is provided for the interpretations to advise each other of their objectives or internally generated goals. Figure 7 depicts the allowed goal transmittals as being the reverse of the allowed action submittals. While this is natural, other possibilities are not to be dismissed.

IV. Testbed

A testbed is currently being developed to resolve issues concerning the CDE methodology. It uses a single simple programming problem, which is discussed in detail in [15].

Analysis of Operating Characteristics

Developing an application programmer's toolkit and a reliable parameterizable arbitrator involves addressing several issues. Four of the problem scenarios that must be overcome are discussed in the following.

Rapid Interpretation. Suppose that an interpretation, I_3 in Figure 7, is coded such that it submits actions much more frequently than the other interpretations—perhaps it does not think very deeply. This induces a performance drain on its acceptance tester, I_2. For an incremental planning problem, the rapid interpretation might dominate the actions taken with its continual minor improvements. The other interpretations could be shut out, because with each action taken they must incorporate the changed plan before proposing any action. The effect is to reduce the module's effectiveness to that of the rapid interpretation, which (under the assumption that its speed comes from a lack of sophistication) may result in the module inadequately responding to changes in the physical environment.

Smooth Decision Boundary. Suppose that an event is becoming more certain (e.g., the approach of an obstacle), for which, an action (EVADE) would be appropriate. The interpretations, in some order, arrive at the conclusion that the same action is advantageous. Suppose that order in Figure 7 is I_3, I_2, and then I_1. The following scenario would ensue: 1) I_3 submits the action to I_2, but it is rejected; 2) I_2 submits the action to I_1, but it is rejected; 3) I_1 submits the action to I_3 and it is approved. The action is not approved until all three interpretations in our example realize its appropriateness. Half the possible orderings result in waiting until the third such realization of the action and half the possible orderings result in waiting only until the second such realization. This problem can be solved by having each interpretation maintain a history of recent actions submitted to it, but that entails a performance penalty. Additionally, a software error, say in I_1, could cause the action to be delayed until I_3 deems it appropriate to resubmit it. Maintaining a history of recent actions submitted also handles the case where there is a software error.

State Corruption. A software error may lead to corruption of the state of an interpretation; e.g., an interpretation's representation of the current plan. Then the module's function becomes vulnerable to an additional software error until the corrupting error is detected and recovery is completed. As discussed earlier, the arbitrator can logically decide to inject a (recovery) exception into the execution of an interpretation. During recovery the system remains vulnerable to an additional software error. If that period of vulnerability is potentially too broad to meet the system reliability requirements, then the interaction between interpretations in Figure 7 could be extended to include all approval pairs or it could be extended to a cycle on four interpretations—as well as more elaborate options.

Hint or Beg. The interpretations have the privilege of transmitting their internally computed goals to each other. This can help offset the problem of infrequent approvals when the interpretations are rather diverse. Suppose, as in Figure 7, that the interpretations are constrained to send goals only to those interpretations that they conduct acceptance tests for. The two natural protocols are when (beg) the arbitrator, perceiving an inadequate acceptance ratio, signals a testee to describe to one of its testers the type of advice it could use and when (hint) the arbitrator, perceiving an inadequate acceptance ratio, signals a tester to provide advice to one of its testees. Hint appears more sensible since, on conducting an acceptance test, the tester could identify a goal (or goals) that is markedly unmet; whereas, it is harder to conceive of the testee formulating a request for the type of advice it needs. It is conceivable, however, that the testing interpretation may be implemented in such a way that it is difficult to determine a critical goal. In that event, it might be better to leave this matter in the testee's demesne.

V. Conclusions

Multiple version software is used to achieve highly reliable software for critical systems, such as for the Airbus slats and flaps. Classical multiple version software models are constrained by the requirement to implement voting, or a similar mechanism, on the software outputs. This constraint limits the degree of diversity allowed the independent programming teams, because their algorithms must reconverge to points of comparison. This thwarts attempts to limit the extent of the unreplicated design/specification phase of the software development process, which constitutes a single point of failure, and constrains (limits diversity of) the independent algorithms.

The proposed methodology uses a test-and-accept mechanism to bypass voting. An effect of the mechanism is to relax the necessity that the requirements and high-level specification be validated for consistency (realizability). Also, conflicts in the human knowledge base that the programming teams access do not need to be resolved before multiple version development begins. For programming projects where human understanding of the problem is imperfect, differences in the opinions of human experts can be resolved during the test-and-accept process. This may have advantages versus requiring (possibly incorrect) decisions to make the human knowledge base consistent in advance.

The methodology relies on back-to-back testing of multiple versions of each module during the iterative development phase. The operational implementation relies on mapping two or more interpretations of each module, plus an arbitrator to a multiprocessor or distributed processing architecture. Handling of hardware failures is not directly considered; it is assumed that a separate methodology, such as SIFT, is used to protect against hardware and communication failures.

The arbitrator for each module is the key element in the software architecture. The arbitrator is a potential single point of failure. Fortunately, the implementation of the arbitrator consists almost entirely of reusable code—it is essentially independent of the local message specification (which describes the format of information exchanged between interpretations).

The interpretations follow an action-event model. Each interpretation, during the course of its computations, arrives at actions that it believes to be advantageous. Here, action may mean a scalar action (individual actions) or a change to the current plan (incremental planning). Proposed actions are submitted to the arbitrator, which chooses one or more other interpretations to examine the proposed action. Based on the acceptance testing of proposed actions, the arbitrator decides which actions are approved and suitably notifies the interpretations so that they may maintain consistent states. The approved actions released by the arbitrator are events perceived by other modules and/or they are system outputs.

The tunable parameters of the arbitrator include the bases for: 1) selecting acceptance testers (also possibly the spawning of interpretations); 2) approval/disapproval of proposed actions; and 3) determining whether to inject an exception into an interpretation to invoke a recovery procedure (acceptance testing results indicating that the interpretation has a corrupted state or other major problem).

Plausible uses of CDE include: 1) operational software that tolerates coding/specification errors for knowledge based systems; 2) operational software that tolerates knowledge base inconsistencies in expert systems; 3) test software to discern inconsistencies in knowledge bases; and 4) test software to identify coding/specification errors for knowledge based systems (like back-to-back testing).

BIBLIOGRAPHY

[1] J. Atlee and J. Gannon, 'State-based model checking of event-driven system requirements,' *IEEE Transactions on Software Engineering,* vol. 19, no. 1, January 1993, pp. 22–40.

[2] A. Avizienis and J. Kelly, 'Fault tolerance by design diversity: Concepts and experiments,' *Computer,* vol. 17, no. 8, August 1984, pp. 67–80.

[3] P. Bishop, et alii, 'Project on Diverse Software—An Experiment in Reliable Software,' *IFAC Workshop SAFECOMP,* October 1985.

[4] L. Chen and A. Avizienis, 'N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation,' *8th Fault Tolerant Computing Symposium,* Toulouse, pp. 3–9, June 1978.

[5] A. Giacalone, P. Mishra, and S. Prasad, 'Facile: A symmetric integration of concurrent and functional programming,' *International Journal of Parallel Programming,* vol. 18, no. 2, 1989.

[6] J. Goodenough and S. Gerhart, 'Toward a theory of test data selection,' *IEEE Transactions on Software Engineering,* vol. 1, no. 2, June 1975.

[7] M. Hecht, K. Tso, and S. Hochhauser, 'The Enhanced Condition Table Methodology for Verification of Critical Software in Ada and C,' *7th International Conference on Testing Computer Software,* San Francisco, June 1990.

[8] A. Hopkins, et alii, 'FTMP—A highly reliable fault tolerant multiprocessor for aircraft.' *Proceedings of the IEEE,* vol. 66, no. 10, October 1978, pp. 1221–1239.

[9] J. Kelly, 'Software Design Diversity,' in *Dependability of Resilient Computers,* (T. Anderson, editor), BSP Professional Books, Oxford, 1989.

[10] J. Kelly and S. Murphy, 'Achieving dependability throughout the development process: A distributed software experiment,' *IEEE Transactions on Software Engineering,* vol. 16, no. 2, February 1990.

[11] K. Kim and H. Welch, 'The distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time applications,' *IEEE Transactions on Computers,* vol. 38, no. 5, May 1989, pp. 626–636.

[12] L. Lamport, R. Shostak, and M. Pease, 'The Byzantine generals problem,' *ACM Transactions on Programming Languages and Systems,* vol. 4, July 1982, pp. 382–401.

[13] F. Meyer and D. Pradhan, 'Dynamic testing strategy for distributed systems,' *IEEE Transactions on Computers,* vol. 38, no. 3, March 1989, pp. 356–365.

[14] F. Meyer and D. Pradhan, 'Consensus with dual failure modes,' *IEEE Transactions on Parallel and Distributed Systems,* vol. 2, no. 2, April 1991, pp. 214–222.

[15] F. Meyer and D. Pradhan, 'A Testbed to Identify Issues in the Cooperating Diverse Experts Methodology for Resolving Inconsistencies in Knowledge Bases,' Technical Report, Texas A&M University Computer Science, (in preparation).

[16] L. Miller, J. Hayes, and S. Mirsky, 'Guidelines for the Verification and Validation of Artificial Intelligence Software Systems,' (preliminary document) Science Applications International Corporation, (for NRC and EPRI), September 1993.

[17] D. Pradhan and N. Vaidya, 'Roll-forward checkpointing scheme: A novel fault-tolerant architecture,' *IEEE Transactions on Computers,* vol. 43, no. 10, October 1994, pp. 1163–1174.

[18] A. Preece, 'Towards a methodology for evaluating expert systems,' *Expert Systems,* vol. 7, no. 4, 1990, pp. 215–223.

[19] F. Preparata, G. Metze, and R. Chien, 'On the connection assignment problem of diagnosable systems,' *IEEE Transactions on Electronic Computing,* vol. 16, December 1967, pp. 848–854.

[20] B. Randell, 'System structure for software fault tolerance,' *IEEE Transactions on Software Engineering,* vol. 1, no. 1, June 1975, pp. 220–232.

[21] D. Smith, 'KIDS: A semi-automatic program development system,' *IEEE Transactions on Software Engineering,* vol. 16, no. 9, September 1990, pp. 1024–1043.

[22] R. Stachowitz and J. Combs, 'Validation of Expert Systems,' *Hawaii International Conference on System Sciences,* January 1987.

[23] N. Vaidya and D. Pradhan, 'Safe system level diagnosis,' *IEEE Transactions on Computers,* vol. 43, March 1994, pp. 367–370.

[24] N. Vaidya and D. Pradhan, 'Degradable Byzantine agreement,' *IEEE Transactions on Computers,* (to appear).

[25] N. Vaidya, A. Singh, and C. Krishna, 'Trade-offs in developing fault tolerant software,' *IEE Proceedings (Part E) Computers and Digital Techniques,* Nov. 1993, pp. 320–326.

[26] J. Wensley, et alii, 'SIFT: The design and analysis of a fault tolerant computer for aircraft control,' *Proceedings of the IEEE,* vol. 66, no. 10, October 1978, pp. 1040–1054.